

8.0

*IBM MQ Developing Applications
Reference*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 1983](#).

This edition applies to version 8 release 0 of IBM® MQ and to all subsequent releases and modifications until otherwise indicated in new editions.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 2007, 2025.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Developing applications reference.....	7
MQI applications reference.....	7
Code examples.....	8
Constants.....	59
Data types used in the MQI.....	231
Function calls.....	598
Attributes of objects.....	768
Return codes.....	838
Rules for validating MQI options.....	839
Queued publish/subscribe command messages.....	842
Machine encodings.....	863
Report options and message flags.....	866
Data-conversion exit.....	870
Properties specified as MQRFH2 elements.....	892
Code page conversion.....	900
Coding standards on 64-bit platforms.....	954
IBM i Application Programming Reference (ILE/RPG).....	958
Data type descriptions.....	958
Function calls.....	1206
Attributes of objects.....	1324
Applications.....	1369
Return codes for IBM i (ILE RPG).....	1381
Rules for validating MQI options for IBM i (ILE RPG).....	1382
Machine encodings on IBM i.....	1385
Report options and message flags.....	1388
Data conversion.....	1391
Conversion processing.....	1392
Processing conventions.....	1393
Conversion of report messages.....	1397
MQDXP - Data-conversion exit parameter.....	1398
MQXCNCV - Convert characters.....	1402
MQCONVX - Data conversion exit.....	1407
SOAP reference.....	1411
amqwSOAPNETListener: IBM MQ SOAP listener for .NET Framework 1 or 2.....	1411
amqswsdl : generate WSDL for .NET Framework 1 or 2 service.....	1413
amqwclientconfig : create Axis 1.4 Web services client deployment descriptor for IBM MQ transport for SOAP.....	1414
amqwdeployWMQService : deploy Web service utility.....	1414
amqwRegisterdotNet : register IBM MQ transport for SOAP to .NET.....	1421
Apache software license.....	1422
MQMD SOAP settings.....	1425
MQRFH2 SOAP settings.....	1431
runivt : IBM MQ transport for SOAP installation verification test.....	1433
Secure Web services over IBM MQ transport for SOAP.....	1435
SimpleJavaListener : IBM MQ SOAP Listener for Axis 1.4.....	1438
IBM MQ SOAP listeners.....	1441
IBM MQ transport for SOAP sender.....	1445
Transactions.....	1446
URI syntax and parameters for Web service deployment.....	1447
W3C SOAP over JMS URI for the IBM MQ Axis 2 client.....	1453
Supported Web services.....	1459
IBM MQ transport for SOAP Web service clients.....	1462


User exits, API exits, and installable services reference.....	1464
MQIEP structure.....	1464
Data-conversion exit reference.....	1468
MQ_PUBLISH_EXIT - Publish exit.....	1472
Channel-exit calls and data structures.....	1480
Cluster workload exit call and data structures.....	1541
API exit reference.....	1566
Installable services interface reference information.....	1624
Installable services interface reference information.....	1686
Reference material for IBM MQ bridge for HTTP.....	1725
HTTP DELETE : IBM MQ bridge for HTTP command.....	1725
HTTP GET : IBM MQ bridge for HTTP command.....	1728
HTTP POST : IBM MQ bridge for HTTP command.....	1731
HTTP headers.....	1734
HTTP return codes.....	1749
Message types and message mappings for WebSphere Bridge for HTTP.....	1757
URI Format.....	1759
The IBM MQ .NET classes and interfaces.....	1760
MQAsyncStatus.NET class.....	1760
MQAuthenticationInformationRecord.NET class.....	1761
MQDestination.NET class.....	1762
MQEnvironment.NET class.....	1764
MQException.NET class.....	1767
MQGetMessageOptions.NET class.....	1767
MQManagedObject.NET class.....	1770
MQMessage.NET class.....	1773
MQProcess.NET class.....	1784
MQPropertyDescriptor.NET class.....	1786
MQPutMessageOptions.NET class.....	1788
MQQueue.NET class.....	1790
MQQueueManager.NET class.....	1798
MQSubscription.NET class.....	1810
MQTopic.NET class.....	1811
IMQObjectTrigger.NET interface.....	1817
MQC.NET interface.....	1818
Character set identifiers for .NET applications.....	1818
IBM MQ C++ classes.....	1820
C++ and MQI cross-reference.....	1822
ImqAuthenticationRecord C++ class.....	1837
ImqBinary C++ class.....	1839
ImqCache C++ class.....	1841
ImqChannel C++ class.....	1844
ImqCICSBridgeHeader C++ class.....	1849
ImqDeadLetterHeader C++ class.....	1855
ImqDistributionList C++ class.....	1858
ImqError C++ class.....	1859
ImqGetMessageOptions C++ class.....	1860
ImqHeader C++ class.....	1864
ImqIMSBridgeHeader C++ class.....	1865
ImqItem C++ class.....	1868
ImqMessage C++ class.....	1870
ImqMessageTracker C++ class.....	1876
ImqNamelist C++ class.....	1879
ImqObject C++ class.....	1880
ImqProcess C++ class.....	1886
ImqPutMessageOptions C++ class.....	1887
ImqQueue C++ class.....	1890
ImqQueueManager C++ class.....	1900

ImqReferenceHeader C++ class.....	1915
ImqString C++ class.....	1918
ImqTrigger C++ class.....	1923
ImqWorkHeader C++ class.....	1926
Properties of IBM MQ classes for JMS objects.....	1928
Dependencies between properties of IBM MQ classes for JMS objects.....	1931
APPLICATIONNAME.....	1933
ASYNCEXCEPTION.....	1933
BROKERCCDURSUBQ.....	1934
BROKERCCSUBQ.....	1935
BROKERCONQ.....	1935
BROKERDURSUBQ.....	1936
BROKERPUBQ.....	1936
BROKERPUBQMGR.....	1937
BROKERQMGR.....	1937
BROKERSUBQ.....	1937
BROKERVER.....	1938
CCDTURL.....	1939
CCSID.....	1939
CHANNEL.....	1940
CLEANUP.....	1940
CLEANUPINT.....	1941
CONNECTIONNAMELIST.....	1941
CLIENTRECONNECTOPTIONS.....	1941
CLIENTRECONNECTTIMEOUT.....	1942
CLIENTID.....	1943
CLONESUPP.....	1943
COMPHDR.....	1944
COMPMSG.....	1944
CONNOPT.....	1945
CONNTAG.....	1946
DESCRIPTION.....	1946
DIRECTAUTH.....	1947
ENCODING.....	1947
EXPIRY.....	1948
FAILIFQUIESCE.....	1948
HOSTNAME.....	1949
LOCALADDRESS.....	1950
MAPNAMESTYLE.....	1950
MAXBUFFSIZE.....	1951
MDREAD.....	1951
MDWRITE.....	1952
MDMSGCTX.....	1952
MSGBATCHSZ.....	1953
MSGBODY.....	1953
MSGRETENTION.....	1954
MSGSELECTION.....	1954
MULTICAST.....	1955
OPTIMISTICPUBLICATION.....	1956
OUTCOMENOTIFICATION.....	1956
PERSISTENCE.....	1957
POLLINGINT.....	1957
PORT.....	1958
PRIORITY.....	1958
PROCESSDURATION.....	1959
PROVIDERVERSION.....	1959
PROXYHOSTNAME.....	1962
PROXYPORT.....	1962

PUBACKINT.....	1963
PUTASYNCALLOWED.....	1963
QMANAGER.....	1964
QUEUE.....	1964
READAHEADALLOWED.....	1965
READAHEADCLOSEPOLICY.....	1965
RECEIVECCSID.....	1966
RECEIVECONVERSION.....	1966
RECEIVEISOLATION.....	1967
RECEXIT.....	1967
RECEXITINIT.....	1968
REPLYTOSTYLE.....	1968
RESCANINT.....	1969
SECEXIT.....	1969
SECEXITINIT.....	1970
SENDCHECKCOUNT.....	1970
SENDEXIT.....	1971
SENDEXITINIT.....	1971
SHARECONVALLOWED.....	1972
SPARSESUBS.....	1972
SSLCIPHERSUITE.....	1973
SSLCRL.....	1973
SSLFIPSREQUIRED.....	1974
SSLPEERNAME.....	1974
SSLRESETCOUNT.....	1975
STATREFRESHINT.....	1975
SUBSTORE.....	1976
SYNCPOINTALLGETS.....	1976
TARGCLIENT.....	1977
TARGCLIENTMATCHING.....	1977
TEMPMODEL.....	1978
TEMPQPREFIX.....	1978
TEMPTOPICPREFIX.....	1979
TOPIC.....	1979
TRANSPORT.....	1979
WILDCARDFORMAT.....	1980
The ENCODING property.....	1981
SSL properties of JMS objects.....	1981
Notices.....	1983
Programming interface information.....	1984
Trademarks.....	1984

Developing applications reference

Use the links provided in this section to help you develop your IBM MQ applications.

- [“MQI applications reference” on page 7](#)
- [“User exits, API exits, and installable services reference” on page 1464](#)
-  [“IBM i Application Programming Reference \(ILE/RPG\)” on page 958](#)
- [“SOAP reference” on page 1411](#)
- [“Reference material for IBM MQ bridge for HTTP” on page 1725](#)
- [“The IBM MQ .NET classes and interfaces” on page 1760](#)
- [“IBM MQ C++ classes” on page 1820](#)
- [The IBM MQ classes for Java libraries](#)
- [IBM MQ classes for JMS](#)

Related tasks

[Developing applications](#)

MQI applications reference

Use the links provided in this section to help you develop your Message Queue Interface (MQI) applications.

- [“Code examples” on page 8](#)
- [“Constants” on page 59](#)
- [“Data types used in the MQI” on page 231](#)
- [“Function calls” on page 598](#)
- [“Attributes of objects” on page 768](#)
- [“Return codes” on page 838](#)
- [“Rules for validating MQI options” on page 839](#)
- [“Machine encodings” on page 863](#)
- [“Report options and message flags” on page 866](#)
- [“Data-conversion exit” on page 870](#)
- [“Properties specified as MQRFH2 elements” on page 892](#)
- [“Code page conversion” on page 900](#)

Related concepts

[“User exits, API exits, and installable services reference” on page 1464](#)

Use the links provided in this section to help you develop your User exits, API exits, and installable services applications:

Related tasks

[Developing applications](#)

Related reference

[“SOAP reference” on page 1411](#)

IBM MQ transport for SOAP reference information arranged alphabetically.

[“Reference material for IBM MQ bridge for HTTP” on page 1725](#)

Reference topics for IBM MQ bridge for HTTP, arranged alphabetically

[“The IBM MQ .NET classes and interfaces” on page 1760](#)

IBM MQ .NET classes and interfaces are listed alphabetically. The properties, methods and constructors are described.

[“IBM MQ C++ classes” on page 1820](#)

The IBM MQ C++ classes encapsulate the IBM MQ Message Queue Interface (MQI). There is a single C++ header file, **imqi.hpp**, which covers all of these classes.

[The IBM MQ Classes for Java libraries](#)

Related information

[../com.ibm.mq.javadoc.doc/WMQJMSClasses/index.html](#)

Code examples

Use the reference information in this section to accomplish the tasks that address your business needs.

C language examples

This collection of topics is mostly taken from the IBM MQ for z/OS sample applications. They are applicable to all platforms, except where noted.

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in z/OS batch.

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
#include <cmqc.h>
...
static char Parm1[MQ_Q_MGR_NAME_LENGTH] ;

int main(int argc, char *argv[] )
{
    /*
    /*      Variables for MQ calls
    /*
    /*      MQHCONN Hconn;      /* Connection handle
    /*      MQLONG  CompCode;    /* Completion code
    /*      MQLONG  Reason;      /* Qualifying reason

    /* Copy the queue manager name, passed in the
    /* parm field, to Parm1
    strncpy(Parm1,argv[1],MQ_Q_MGR_NAME_LENGTH);

    /*
    /* Connect to the specified queue manager.
    /* Test the output of the connect call. If the
    /* call fails, print an error message showing the
    /* completion code and reason code, then leave the
    /* program.
    /*
    MQCONN(Parm1,
           &Hconn,
           &CompCode,
           &Reason);
    if ((CompCode != MQCC_OK) | (Reason != MQRC_NONE))
    {
        sprintf(pBuff, MESSAGE_4_E,
                ERROR_IN_MQCONN, CompCode, Reason);
        PrintLine(pBuff);
        RetCode = CSQ4_ERROR;
        goto AbnormalExit2;
    }
    ...
}
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in z/OS batch.

The variables used in this code extract are those that were set in “Connecting to a queue manager” on page 8. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
:
/*
/* Disconnect from the queue manager. Test the
/* output of the disconnect call. If the call
/* fails, print an error message showing the
/* completion code and reason code.
/*
MQDISC(&Hconn,
      &CompCode,
      &Reason);
if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQDISC, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
:
```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue.

This extract is taken from the Mail Manager sample application (program CSQ4TCD1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
:
MQLONG HCONN = 0; /* Connection handle */
MQHOBJS HOBJ; /* MailQ Object handle */
MQHOBJS HobjTempQ; /* TempQ Object Handle */
MQLONG CompCode; /* Completion code */
MQLONG Reason; /* Qualifying reason */
MQOD ObjDesc = {MQOD_DEFAULT}; /* Object descriptor */
MQLONG OpenOptions; /* Options control MQOPEN */

/*-----*/
/* Initialize the Object Descriptor (MQOD) */
/* control block. (The remaining fields */
/* are already initialized.) */
/*-----*/
strncpy( ObjDesc.ObjectName,
        SYSTEM_REPLY_MODEL,
        MQ_Q_NAME_LENGTH );
strncpy( ObjDesc.DynamicQName,
        SYSTEM_REPLY_INITIAL,
        MQ_Q_NAME_LENGTH );
OpenOptions = MQOO_INPUT_AS_Q_DEF;
/*-----*/
/* Open the model queue and, therefore, */
/* create and open a temporary dynamic */
/* queue */
/*-----*/
MQOPEN( HCONN,
        &ObjDesc,
        OpenOptions,
        &HobjTempQ,
        &CompCode,
        &Reason );
if ( CompCode == MQCC_OK ) {
}
else {
    /*-----*/
    /* Build an error message to report the */
    /* failure of the opening of the model */
    /* -----*/
}
```

```

/* queue */
/*-----*/
MQMErrorHandling( "OPEN TEMPQ", CompCode,
                  Reason );
    ErrorFound = TRUE;
}
return ErrorFound;
}
...

```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open a queue that has already been defined.

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

#include <cmqc.h>
...
static char Parm1[MQ_Q_MGR_NAME_LENGTH];
...
int main(int argc, char *argv[] )
{
/*
/*      Variables for MQ calls                                */
/*
MQHCONN Hconn ;          /* Connection handle                */
MQLONG  CompCode;        /* Completion code      */
MQLONG  Reason;          /* Qualifying reason     */
MQOD     ObjDesc = { MQOD_DEFAULT };
MQLONG  OpenOptions;     /* Object descriptor     */
/* Options that control */
/* the MQOPEN call      */
MQHOBJ   Hobj;           /* Object handle         */
...
/* Copy the queue name, passed in the parm field, */
/* to Parm2 strncpy(Parm2,argv[2],                */
/* MQ_Q_NAME_LENGTH);                             */
...
/*
/* Initialize the object descriptor (MQOD) control */
/* block. (The initialization default sets StrucId, */
/* Version, ObjectType, ObjectQMgrName,           */
/* DynamicQName, and AlternateUserid fields)      */
/*
strncpy(ObjDesc.ObjectName,Parm2,MQ_Q_NAME_LENGTH);
...
/* Initialize the other fields required for the open */
/* call (Hobj is set by the MQCONN call).            */
/*
OpenOptions = MQOO_BROWSE;
...
/*
/* Open the queue.                                         */
/* Test the output of the open call. If the call        */
/* fails, print an error message showing the            */
/* completion code and reason code, then bypass        */
/* processing, disconnect and leave the program.        */
/*
MQOPEN(Hconn,
        &ObjDesc,
        OpenOptions,
        &Hobj,
        &CompCode,
        &Reason);

if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQOPEN, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
    goto AbnormalExit1;    /* disconnect processing */
}
...
} /* end of main */

```

Closing a queue


This example demonstrates how to use the MQCLOSE call to close a queue.

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
:
/*                                     */
/* Close the queue.                   */
/* Test the output of the close call. If the call */
/* fails, print an error message showing the */
/* completion code and reason code.         */
/*                                     */
MQCLOSE(Hconn,
        &Hobj,
        MQCO_NONE,
        &CompCode,
        &Reason);
if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQCLOSE, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
:
```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call to put a message on a queue.

This extract is not taken from the sample applications supplied with IBM MQ. For the names and locations of the sample applications, see [Sample procedural programs \(platforms except z/OS\)](#)  and [Sample programs for IBM MQ for z/OS](#).

```
:
qput()
{
    MQMD    MsgDesc;
    MQPMO    PutMsgOpts;
    MQLONG    CompCode;
    MQLONG    Reason;
    MQHCONN    Hconn;
    MQHOBJ    Hobj;
    char message_buffer[] = "MY MESSAGE";
    /*-----*/
    /* Set up PMO structure. */
    /*-----*/
    memset(&PutMsgOpts, '\0', sizeof(PutMsgOpts));
    memcpy(PutMsgOpts.StrucId, MQPMO_STRUC_ID,
           sizeof(PutMsgOpts.StrucId));
    PutMsgOpts.Version = MQPMO_VERSION_1;
    PutMsgOpts.Options = MQPMO_SYNCPOINT;

    /*-----*/
    /* Set up MD structure. */
    /*-----*/
    memset(&MsgDesc, '\0', sizeof(MsgDesc));
    memcpy(MsgDesc.StrucId, MQMD_STRUC_ID,
           sizeof(MsgDesc.StrucId));
    MsgDesc.Version = MQMD_VERSION_1;
    MsgDesc.Expiry = MQEI_UNLIMITED;
    MsgDesc.Report = MQRO_NONE;
    MsgDesc.MsgType = MQMT_DATAGRAM;
    MsgDesc.Priority = 1;
    MsgDesc.Persistence = MQPER_PERSISTENT;
    memset(MsgDesc.ReplyToQ,
           '\0',
           sizeof(MsgDesc.ReplyToQ));
    /*-----*/
    /* Put the message. */
    /*-----*/
    MQPUT(Hconn, Hobj, &MsgDesc, &PutMsgOpts,
```

```

        sizeof(message_buffer), message_buffer,
        &CompCode, &Reason);

/*-----*/
/* Check completion and reason codes. */
/*-----*/
switch (CompCode)
{
    case MQCC_OK:
        break;
    case MQCC_FAILED:
        switch (Reason)
        {
            case MQRC_Q_FULL:
            case MQRC_MSG_TOO_BIG_FOR_Q:
                break;
            default:
                break; /* Perform error processing */
        }
        break;
    default:
        break; /* Perform error processing */
}
}

```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call to open a queue, put a single message on the queue, then close the queue.

This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
MQLONG  Hconn;           /* Connection handle      */
MQHOBJ  Hobj_CheckQ;     /* Object handle         */
MQLONG  CompCode;        /* Completion code       */
MQLONG  Reason;          /* Qualifying reason     */
MQOD    ObjDesc = {MQOD_DEFAULT}; /* Object descriptor    */
MQMD    MsgDesc = {MQMD_DEFAULT}; /* Message descriptor   */
MQLONG  OpenOptions;     /* Control the MQOPEN call */
MQGMO    GetMsgOpts = {MQGMO_DEFAULT}; /* Get Message Options */
MQLONG  MsgBuffLen;      /* Length of message buffer */
CSQ4BCAQ MsgBuffer;      /* Message structure     */
MQLONG  DataLen;         /* Length of message     */

MQPMO    PutMsgOpts = {MQPMO_DEFAULT}; /* Put Message Options */
CSQ4BQRM PutBuffer;      /* Message structure     */
MQLONG  PutBuffLen = sizeof(PutBuffer); /* Length of message buffer */
:

```

```

void Process_Query(void)
{
    /*
    /* Build the reply message
    /*
    :
    /*
    /* Set the object descriptor, message descriptor and
    /* put message options to the values required to
    /* create the reply message.
    /*
    strncpy(ObjDesc.ObjectName, MsgDesc.ReplyToQ,
            MQ_Q_NAME_LENGTH);
    strncpy(ObjDesc.ObjectQMGrName, MsgDesc.ReplyToQMGr,
            MQ_Q_MGR_NAME_LENGTH);
    MsgDesc.MsgType = MQMT_REPLY;
    MsgDesc.Report = MQRO_NONE;
    memset(MsgDesc.ReplyToQ, ' ', MQ_Q_NAME_LENGTH);
    memset(MsgDesc.ReplyToQMGr, ' ', MQ_Q_MGR_NAME_LENGTH);

```

```

memcpy(MsgDesc.MsgId, MQMI_NONE, sizeof(MsgDesc.MsgId));
PutMsgOpts.Options = MQPMO_SYNCPOINT +
                    MQPMO_PASS_IDENTITY_CONTEXT;
PutMsgOpts.Context = Hobj_CheckQ;
PutBufLen = sizeof(PutBuffer);
MQPUT1(Hconn,
        &ObjDesc,
        &MsgDesc,
        &PutMsgOpts,
        PutBufLen,
        &PutBuffer,
        &CompCode,
        &Reason);

if (CompCode != MQCC_OK)
{
    strncpy(TS_Operation, "MQPUT1",
            sizeof(TS_Operation));
    strncpy(TS_ObjName, ObjDesc.ObjectName,
            MQ_Q_NAME_LENGTH);
    Record_Call_Error();
    Forward_Msg_To_DLQ();
}
return;
}
:

```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue.

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

#include "cmqc.h"
:
#define BUFFERLENGTH 80
:
int main(int argc, char *argv[] )
{
    /*                                     */
    /*      Variables for MQ calls      */
    /*                                     */
    MQHCONN Hconn ;                      /* Connection handle */
    MQLONG  CompCode;                    /* Completion code   */
    MQLONG  Reason;                      /* Qualifying reason */
    MQHOBJ  Hobj;                       /* Object handle     */
    MQMD    MsgDesc = { MQMD_DEFAULT }; /* Message descriptor */
    MQLONG  DataLength ;                 /* Length of the message */
    MQCHAR  Buffer[BUFFERLENGTH+1];      /* Area for message data */
    MQGMO    GetMsgOpts = { MQGMO_DEFAULT }; /* Options which control */
    /*                                     */
    /*                                     */
    /* the MQGET call */
    MQLONG  BufferLength = BUFFERLENGTH ; /* Length of buffer   */
    :
    /*      No need to change the message descriptor */
    /*      (MQMD) control block because initialization */
    /*      default sets all the fields. */
    /*                                     */
    /*      Initialize the get message options (MQGMO) */
    /*      control block (the copy file initializes all */
    /*      the other fields). */
    /*                                     */
    GetMsgOpts.Options = MQGMO_NO_WAIT +
                        MQGMO_BROWSE_FIRST +
                        MQGMO_ACCEPT_TRUNCATED_MSG;

    /*                                     */
    /* Get the first message. */
    /* Test for the output of the call is carried out */
    /* in the 'for' loop. */
    /*                                     */
    MQGET(Hconn,
          Hobj,
          &MsgDesc,
          &GetMsgOpts,

```

```

        BufferLength,
        Buffer,
        &DataLength,
        &CompCode,
        &Reason);

/*
/* Process the message and get the next message,
/* until no messages remaining.
...
/* If the call fails for any other reason,
/* print an error message showing the completion
/* code and reason code.
...
if ( (CompCode == MQCC_FAILED) &&
    (Reason == MQRC_NO_MSG_AVAILABLE) )
{
    ...
}
else
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQGET, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
...
} /* end of main */

```

Getting a message using the wait option

This example demonstrates how to use the wait option of the MQGET call.

This code accepts truncated messages. This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
MQLONG   Hconn;           /* Connection handle      */
MQHOBJ   Hobj_CheckQ;     /* Object handle        */
MQLONG   CompCode;        /* Completion code      */
MQLONG   Reason;          /* Qualifying reason    */
MQOD     ObjDesc = {MQOD_DEFAULT}; /* Object descriptor    */
MQMD     MsgDesc = {MQMD_DEFAULT}; /* Message descriptor    */
MQLONG   OpenOptions;
MQGMO    GetMsgOpts = {MQGMO_DEFAULT}; /* Get Message Options */
MQLONG   MsgBuffLen;      /* Length of message buffer */
CSQ4BCAQ MsgBuffer;       /* Message structure     */
MQLONG   DataLen;         /* Length of message     */

```

```

:
void main(void)
{
    ...
    /*
    /* Initialize options and open the queue for input
    /*
    ...
    /*
    /* Get and process messages
    /*
    GetMsgOpts.Options = MQGMO_WAIT +
                        MQGMO_ACCEPT_TRUNCATED_MSG +
                        MQGMO_SYNCPOINT;
    GetMsgOpts.WaitInterval = WAIT_INTERVAL;
    MsgBuffLen = sizeof(MsgBuffer);
    memcpy(MsgDesc.MsgId, MQMI_NONE,
           sizeof(MsgDesc.MsgId));
    memcpy(MsgDesc.CorrelId, MQCI_NONE,
           sizeof(MsgDesc.CorrelId));
    /*
    /* Make the first MQGET call outside the loop
    /*

```

```

MQGET(Hconn,
      Hobj_CheckQ,
      &MsgDesc,
      &GetMsgOpts,
      MsgBufLen,
      &MsgBuffer,
      &DataLen,
      &CompCode,
      &Reason);
:
/*
/* Test the output of the MQGET call. If the call */
/* failed, send an error message showing the */
/* completion code and reason code, unless the */
/* reason code is NO_MSG_AVAILABLE. */
/* */
if (Reason != MQRC_NO_MSG_AVAILABLE)
{
    strncpy(TS_Operation, "MQGET", sizeof(TS_Operation));
    strncpy(TS_ObjName, ObjDesc.ObjectName,
            MQ_Q_NAME_LENGTH);
    Record_Call_Error();
}
:

```

Getting a message using signaling

Signaling is available only with IBM MQ for z/OS .

This example demonstrates how to use the MQGET call to set a signal so that you are notified when a suitable message arrives on a queue. This extract is not taken from the sample applications supplied with IBM MQ.

```

:
get_set_signal()
{
    MQMD      MsgDesc;
    MQGMO     GetMsgOpts;
    MQLONG    CompCode;
    MQLONG    Reason;
    MQHCONN   Hconn;
    MQHOBJ    Hobj;
    MQLONG    BufferLength;
    MQLONG    DataLength;
    char message_buffer[100];
    long int q_ecb, work_ecb;
    short int signal_sw, endloop;
    long int mask = 255;

    /*-----*/
    /* Set up GMO structure. */
    /*-----*/
    memset(&GetMsgOpts, '\0', sizeof(GetMsgOpts));
    memcpy(GetMsgOpts.StrucId, MQGMO_STRUC_ID,
           sizeof(GetMsgOpts.StrucId));
    GetMsgOpts.Version      = MQGMO_VERSION_1;
    GetMsgOpts.WaitInterval = 1000;
    GetMsgOpts.Options      = MQGMO_SET_SIGNAL +
                              MQGMO_BROWSE_FIRST;

    q_ecb = 0;
    GetMsgOpts.Signal1 = &q_ecb;
    /*-----*/
    /* Set up MD structure. */
    /*-----*/
    memset(&MsgDesc, '\0', sizeof(MsgDesc));
    memcpy(MsgDesc.StrucId, MQMD_STRUC_ID,
           sizeof(MsgDesc.StrucId));
    MsgDesc.Version = MQMD_VERSION_1;
    MsgDesc.Report  = MQRO_NONE;
    memcpy(MsgDesc.MsgId, MQMI_NONE,
           sizeof(MsgDesc.MsgId));
    memcpy(MsgDesc.CorrelId, MQCI_NONE,
           sizeof(MsgDesc.CorrelId));

    /*-----*/
    /* Issue the MQGET call. */
    /*-----*/
    BufferLength = sizeof(message_buffer);

```

```

signal_sw = 0;

MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
      BufferLength, message_buffer, &DataLength,
      &CompCode, &Reason);
/*-----*/
/* Check completion and reason codes. */
/*-----*/
switch (CompCode)
{
    case (MQCC_OK):          /* Message retrieved */
        break;
    case (MQCC_WARNING):
        switch (Reason)
        {
            case (MQRC_SIGNAL_REQUEST_ACCEPTED):
                signal_sw = 1;
                break;
            default:
                break; /* Perform error processing */
        }
        break;
    case (MQCC_FAILED):
        switch (Reason)
        {
            case (MQRC_Q_MGR_NOT_AVAILABLE):
            case (MQRC_CONNECTION_BROKEN):
            case (MQRC_Q_MGR_STOPPING):
                break;
            default:
                break; /* Perform error processing. */
        }
        break;
    default:
        break; /* Perform error processing. */
}

/*-----*/
/* If the SET_SIGNAL was accepted, set up a loop to */
/* check whether a message has arrived at one second */
/* intervals. The loop ends if a message arrives or */
/* the wait interval specified in the MQGMO */
/* structure has expired. */
/*-----*/
/* If a message arrives on the queue, another MQGET */
/* must be issued to retrieve the message. If other */
/* MQM calls have been made in the intervening */
/* period, this may necessitate reinitializing the */
/* MQMD and MQGMO structures. */
/* In this code, no intervening calls */
/* have been made, so the only change required to */
/* the structures is to specify MQGMO_NO_WAIT, */
/* since we now know the message is there. */
/*-----*/
/* This code uses the EXEC CICS DELAY command to */
/* suspend the program for a second. A batch program */
/* may achieve the same effect by calling an */
/* assembler language subroutine which issues a */
/* z/OS STIMER macro. */
/*-----*/

if (signal_sw == 1)
{
    endloop = 0;
    do
    {
        EXEC CICS DELAY FOR HOURS(0) MINUTES(0) SECONDS(1);
        work_ecb = q_ecb & mask;
        switch (work_ecb)
        {
            case (MQEC_MSG_ARRIVED):
                endloop = 1;
                mqgmo_options = MQGMO_NO_WAIT;
                MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
                    BufferLength, message_buffer,
                    &DataLength, &CompCode, &Reason);
                if (CompCode != MQCC_OK)
                {
                    ; /* Perform error processing. */
                }
                break;
            case (MQEC_WAIT_INTERVAL_EXPIRED):
            case (MQEC_WAIT_CANCELED):
                endloop = 1;
        }
    }
}

```

```

        break;
    default:
        break;
    }
} while (endloop == 0);
}
return;
}

```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue.

This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

#include <cmqc.h>      /* MQ API header file      */
...
#define NUMBEROFSELECTORS 2

const MQHCONN Hconn = MQHC_DEF_HCONN;
...
static void InquireGetAndPut(char *Message,
                           PMQHOBJ pHobj,
                           char *Object)

{
    /*      Declare local variables      */
    /*      */
    MQLONG SelectorCount = NUMBEROFSELECTORS;
    /*      Number of selectors      */
    MQLONG IntAttrCount = NUMBEROFSELECTORS;
    /*      Number of int attrs      */
    MQLONG CharAttrLength = 0;
    /*      Length of char attribute buffer      */
    MQCHAR *CharAttrs ;
    /*      Character attribute buffer      */
    MQLONG SelectorsTable[NUMBEROFSELECTORS];
    /*      attribute selectors      */
    MQLONG IntAttrsTable[NUMBEROFSELECTORS];
    /*      integer attributes      */
    MQLONG CompCode;
    /*      Completion code      */
    MQLONG Reason;
    /*      Qualifying reason      */
    /*      */
    /*      Open the queue. If successful, do the inquire */
    /*      call.      */
    /*      */
    /*      Initialize the variables for the inquire */
    /*      call:      */
    /*      - Set SelectorsTable to the attributes whose */
    /*      status is      */
    /*      required      */
    /*      - All other variables are already set      */
    /*      */
    SelectorsTable[0] = MQIA_INHIBIT_GET;
    SelectorsTable[1] = MQIA_INHIBIT_PUT;
    /*      */
    /*      Issue the inquire call      */
    /*      Test the output of the inquire call. If the */
    /*      call failed, display an error message      */
    /*      showing the completion code and reason code, */
    /*      otherwise display the status of the      */
    /*      INHIBIT-GET and INHIBIT-PUT attributes      */
    /*      */
    MQINQ(Hconn,
          *pHobj,
          SelectorCount,
          SelectorsTable,
          IntAttrCount,
          IntAttrsTable,
          CharAttrLength,
          CharAttrs,
          &CompCode,
          &Reason);
    if (CompCode != MQCC_OK)
    {

```

```

        sprintf(Message, MESSAGE_4_E,
                ERROR_IN_MQINQ, CompCode, Reason);
        SetMsg(Message);
    }
    else
    {
        /* Process the changes */
    } /* end if CompCode */

```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue.

This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

#include <cmqc.h>      /* MQ API header file      */
...
#define NUMBEROFSELECTORS 2

const MQHCONN Hconn = MQHC_DEF_HCONN;

static void InhibitGetAndPut(char *Message,
                            PMQHOBJ pHobj,
                            char *Object)
{
    /*
    /*      Declare local variables
    /*
    /*
    MQLONG SelectorCount = NUMBEROFSELECTORS;
    MQLONG IntAttrCount = NUMBEROFSELECTORS;
    MQLONG CharAttrLength = 0;
    MQCHAR *CharAttrs ;
    MQLONG SelectorsTable[NUMBEROFSELECTORS];
    MQLONG IntAttrsTable[NUMBEROFSELECTORS];
    MQLONG CompCode;
    MQLONG Reason;
    :
    /*
    /*      Open the queue. If successful, do the
    /*      inquire call.
    /*
    :
    /*
    /*      Initialize the variables for the set call:
    /*      - Set SelectorsTable to the attributes to be
    /*      set
    /*      - Set IntAttrsTable to the required status
    /*      - All other variables are already set
    /*
    SelectorsTable[0] = MQIA_INHIBIT_GET;
    SelectorsTable[1] = MQIA_INHIBIT_PUT;
    IntAttrsTable[0] = MQQA_GET_INHIBITED;
    IntAttrsTable[1] = MQQA_PUT_INHIBITED;
    :

    /*
    /*      Issue the set call.
    /*      Test the output of the set call. If the
    /*      call fails, display an error message
    /*      showing the completion code and reason
    /*      code; otherwise move INHIBITED to the
    /*      relevant screen map fields
    /*
    MQSET(Hconn,
          *pHobj,
          SelectorCount,
          SelectorsTable,
          IntAttrCount,
          IntAttrsTable,
          CharAttrLength,

```

```

        CharAttrs,
        &CompCode,
        &Reason);
if (CompCode != MQCC_OK)
{
    sprintf(Message, MESSAGE_4_E,
            ERROR_IN_MQSET, CompCode, Reason);
    SetMsg(Message);
}
else
{
    /* Process the changes */
} /* end if CompCode */

```

Retrieving status information with MQSTAT

This example demonstrates how to issue an asynchronous MQPUT and retrieve the status information with MQSTAT.

This extract is taken from the Calling MQSTAT sample application (program amqsapt0) supplied with IBM MQ for Windows systems. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS \)](#).

```

/*****
/*
/* Program name: AMQSAPT0
/*
/* Description: Sample C program that asynchronously puts messages
/* to a message queue (example using MQPUT & MQSTAT).
/*
/*
/* Licensed Materials - Property of IBM
/*
/* 63H9336
/* (c) Copyright IBM Corp. 2006, 2022. All Rights Reserved.
/*
/* US Government Users Restricted Rights - Use, duplication or
/* disclosure restricted by GSA ADP Schedule Contract with
/* IBM Corp.
/*
*****/
/* Function:
/*
/* AMQSAPT0 is a sample C program to put messages on a message
/* queue with asynchronous response option, querying the success
/* of the put operations with MQSTAT.
/*
/* -- messages are sent to the queue named by the parameter
/*
/* -- gets lines from StdIn, and adds each to target
/* queue, taking each line of text as the content
/* of a datagram message; the sample stops when a null
/* line (or EOF) is read.
/* New-line characters are removed.
/* If a line is longer than 99 characters it is broken up
/* into 99-character pieces. Each piece becomes the
/* content of a datagram message.
/* If the length of a line is a multiple of 99 plus 1, for
/* example, 199, the last piece will only contain a
/* new-line character so will terminate the input.
/*
/* -- writes a message for each MQI reason other than
/* MQRC_NONE; stops if there is a MQI completion code
/* of MQCC_FAILED
/*
/* -- summarizes the overall success of the put operations
/* through a call to MQSTAT to query MQSTAT_TYPE_ASYNC_ERROR*/
/*
/* Program logic:
/* MQOPEN target queue for OUTPUT
/* while end of input file not reached,
/* . read next line of text
/* . MQPUT datagram message with text line as data
/* MQCLOSE target queue
/* MQSTAT connection
/*
*****/
/*

```

```

/* AMQSAPTO has the following parameters */
/* required: */
/* (1) The name of the target queue */
/* optional: */
/* (2) Queue manager name */
/* (3) The open options */
/* (4) The close options */
/* (5) The name of the target queue manager */
/* (6) The name of the dynamic queue */
/* */
/*****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* includes for MQI */
#include <cmqc.h>

int main(int argc, char **argv)
{
    /* Declare file and character for sample input */
    FILE *fp;

    /* Declare MQI structures needed */
    MQOD od = {MQOD_DEFAULT}; /* Object Descriptor */
    MQMD md = {MQMD_DEFAULT}; /* Message Descriptor */
    MQPMO pmo = {MQPMO_DEFAULT}; /* put message options */
    MQSTS sts = {MQSTS_DEFAULT}; /* status information */
    /** note, sample uses defaults where it can */
    MQHCONN Hcon; /* connection handle */
    MQHOBJ Hobj; /* object handle */
    MQLONG O_options; /* MQOPEN options */
    MQLONG C_options; /* MQCLOSE options */
    MQLONG CompCode; /* completion code */
    MQLONG OpenCode; /* MQOPEN completion code */
    MQLONG Reason; /* reason code */
    MQLONG CReason; /* reason code for MQCONN */
    MQLONG messlen; /* message length */
    char buffer[100]; /* message buffer */
    char QMName[50]; /* queue manager name */

    printf("Sample AMQSAPTO start\n");
    if (argc < 2)
    {
        printf("Required parameter missing - queue name\n");
        exit(99);
    }

    /*****
    /* Connect to queue manager */
    /* */
    QMName[0] = 0; /* default */
    if (argc > 2)
        strcpy(QMName, argv[2]);
    MQCONN(QMName, /* queue manager */
           &Hcon, /* connection handle */
           &Compcode, /* completion code */
           &Reason); /* reason code */
    /* report reason and stop if it failed */
    if (CompCode == MQCC_FAILED)
    {
        printf("MQCONN ended with reason code %d\n", CReason);
        exit( (int)CReason );
    }

    /*****
    /* Use parameter as the name of the target queue */
    /* */
    strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
    printf("target queue is %s\n", od.ObjectName);

    if (argc > 5)
    {
        strncpy(od.ObjectQMGrName, argv[5], (size_t) MQ_Q_MGR_NAME_LENGTH);
        printf("target queue manager is %s\n", od.ObjectQMGrName);
    }

    if (argc > 6)
    {

```

```

    strncpy(od.DynamicQName, argv[6], (size_t) MQ_Q_NAME_LENGTH);
    printf("dynamic queue name is %s\n", od.DynamicQName);
}

/*****
/*
/*  Open the target message queue for output
/*
/*
*****/
if (argc > 3)
{
    O_options = atoi( argv[3] );
    printf("open options are %d\n", O_options);
}
else
{
    O_options = MQOO_OUTPUT          /* open queue for output      */
                | MQOO_FAIL_IF_QUIESCING /* but not if MQM stopping */
                ;                  /* = 0x2010 = 8208 decimal */
}

MQOPEN(Hcon,                      /* connection handle      */
        &od,                      /* object descriptor for queue */
        O_options,                /* open options           */
        &Hobj,                   /* object handle          */
        &OpenCode,               /* MQOPEN completion code */
        &Reason);               /* reason code            */

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE)
{
    printf("MQOPEN ended with reason code %d\n", Reason);
}

if (OpenCode == MQCC_FAILED)
{
    printf("unable to open queue for output\n");
}

/*****
/*
/*  Read lines from the file and put them to the message queue
/*
/*  Loop until null line or end of file, or there is a failure
/*
*****/
CompCode = OpenCode;          /* use MQOPEN result for initial test */
fp = stdin;

memcpy(md.Format,              /* character string format      */
        MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

/*****
/* These options specify that put operation should occur
/*
/* asynchronously and the application will check the success
/* using MQSTAT at a later time.
*****/
md.Persistence = MQPER_NOT_PERSISTENT;
pmo.Options |= MQPMO_ASYNC_RESPONSE;

/*****
/* These options cause the MsgId and CorrelId to be replaced, so
/* that there is no need to reset them before each MQPUT
*****/
pmo.Options |= MQPMO_NEW_MSG_ID;
pmo.Options |= MQPMO_NEW_CORREL_ID;

while (CompCode != MQCC_FAILED)
{
    if (fgets(buffer, sizeof(buffer), fp) != NULL)
    {
        messlen = (MQLONG)strlen(buffer); /* length without null */
        if (buffer[messlen-1] == '\n') /* last char is a new-line */
        {
            buffer[messlen-1] = '\0'; /* replace new-line with null */
            --messlen; /* reduce buffer length */
        }
    }
    else messlen = 0; /* treat EOF same as null line */

/*****
/*
/*  Put each buffer to the message queue
*****/

```

```

/*
/*****
if (messlen > 0)
{
    MQPUT(Hcon,          /* connection handle          */
          Hobj,          /* object handle          */
          &md,           /* message descriptor     */
          &pmo,          /* default options (datagram) */
          messlen,       /* message length         */
          buffer,        /* message buffer         */
          &CompCode,     /* completion code        */
          &Reason);      /* reason code            */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {
        printf("MQPUT ended with reason code %d\n", Reason);
    }
}
else /* satisfy end condition when empty line is read */
    CompCode = MQCC_FAILED;
}

/*****
/*
/* Close the target queue (if it was opened)
/*
/*****
if (OpenCode != MQCC_FAILED)
{
    if (argc > 4)
    {
        C_options = atoi( argv[4] );
        printf("close options are %d\n", C_options);
    }
    else
    {
        C_options = MQCO_NONE; /* no close options */
    }

    MQCLOSE(Hcon,          /* connection handle          */
            &Hobj,         /* object handle              */
            C_options,     /* completion code            */
            &CompCode,     /* completion code            */
            &Reason);      /* reason code                */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {
        printf("MQCLOSE ended with reason code %d\n", Reason);
    }
}

/*****
/*
/* Query how many asynchronous puts succeeded
/*
/*****
MQSTAT(&Hcon,          /* connection handle          */
       MQSTAT_TYPE_ASYNC_ERROR, /* status type          */
       &Sts,           /* MQSTS structure          */
       &CompCode,      /* completion code          */
       &Reason);       /* reason code              */

/* report reason, if any */
if (Reason != MQRC_NONE)
{
    printf("MQSTAT ended with reason code %d\n", Reason);
}
else
{
    /* Display results */
    printf("Succeeded putting %d messages\n",
           sts.PutSuccessCount);
    printf("%d messages were put with a warning\n",
           sts.PutWarningCount);
    printf("Failed to put %d messages\n",
           sts.PutFailureCount);

    if (sts.CompCode == MQCC_WARNING)
    {
        printf("The first warning that occurred had reason code %d\n",

```

```

        sts.Reason);
    }
    else if(sts.CompCode == MQCC_FAILED)
    {
        printf("The first error that occurred had reason code %d\n",
            sts.Reason);
    }
}

/*****
/*
/*   Disconnect from MQM if not already connected
/*
/*
/* *****/
if (CReason != MQRC_ALREADY_CONNECTED)
{
    MQDISC(&Hcon,          /* connection handle      */
          &CompCode,      /* completion code       */
          &Reason);       /* reason code            */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {
        printf("MQDISC ended with reason code %d\n", Reason);
    }
}

/*****
/*
/*   END OF AMQSAPT0
/*
/* *****/
printf("Sample AMQSAPT0 end\n");
return(0);
}

```

COBOL examples

This collection of topics is taken from the IBM MQ for z/OS sample applications. They are applicable to all platforms, except where noted.

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in z/OS batch.

This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

* -----*
WORKING-STORAGE SECTION.
* -----*
*   W02 - Data fields derived from the PARM field
01 W02-MQM          PIC X(48) VALUE SPACES.
*   W03 - MQM API fields
01 W03-HCONN        PIC S9(9) BINARY.
01 W03-COMPCODE     PIC S9(9) BINARY.
01 W03-REASON       PIC S9(9) BINARY.
*
*   MQV contains constants (for filling in the control
*   blocks)
*   and return codes (for testing the result of a call)
*
01 W05-MQM-CONSTANTS.
COPY CMQV SUPPRESS.
:
*   Separate into the relevant fields any data passed
*   in the PARM statement
*
UNSTRING PARM-STRING DELIMITED BY ALL ','
          INTO W02-MQM
          W02-OBJECT.
:
*   Connect to the specified queue manager.
*
CALL 'MQCONN' USING W02-MQM

```

```

                                W03-HCONN
                                W03-COMPCODE
                                W03-REASON.
*
*   Test the output of the connect call.  If the call
*   fails, print an error message showing the
*   completion code and reason code.
*
*   IF (W03-COMPCODE NOT = MQCC-OK) THEN
*   ..
*   END-IF.
*   ..

```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in z/OS batch.

The variables used in this code extract are those that were set in [“Connecting to a queue manager”](#) on page 23. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
*
* Disconnect from the queue manager
*
*   CALL 'MQDISC' USING W03-HCONN
*                       W03-COMPCODE
*                       W03-REASON.
*
*   Test the output of the disconnect call.  If the
*   call fails, print an error message showing the
*   completion code and reason code.
*
*   IF (W03-COMPCODE NOT = MQCC-OK) THEN
*   ..
*   END-IF.
*   ..

```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue.

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W02 - Queues processed in this program
*
*01 W02-MODEL-QNAME          PIC X(48) VALUE
   'CSQ4SAMP.B1.MODEL          '
*01 W02-NAME-PREFIX         PIC X(48) VALUE
   'CSQ4SAMP.B1.*              '
*01 W02-TEMPORARY-Q         PIC X(48).
*
*   W03 - MQM API fields
*
*01 W03-HCONN              PIC S9(9) BINARY VALUE ZERO.
*01 W03-OPTIONS            PIC S9(9) BINARY.
*01 W03-HOBJ               PIC S9(9) BINARY.
*01 W03-COMPCODE           PIC S9(9) BINARY.
*01 W03-REASON             PIC S9(9) BINARY.
*
*   API control blocks
*
*01 MQM-OBJECT-DESCRIPTOR.
   COPY CMQODV.
*
*   CMQV contains constants (for setting or testing
*   field values) and return codes (for testing the

```

```

*   result of a call)
*
*   01 MQM-CONSTANTS.
*   COPY CMQV SUPPRESS.
* -----*
*   PROCEDURE DIVISION.
* -----*
*   :
* -----*
*   OPEN-TEMP-RESPONSE-QUEUE SECTION.
* -----*

```

```

*
*   This section creates a temporary dynamic queue
*   using a model queue
*
* -----*
*
*   Change three fields in the Object Descriptor (MQOD)
*   control block. (MQODV initializes the other fields)
*
*   MOVE MQOT-Q           TO MQOD-OBJECTTYPE.
*   MOVE W02-MODEL-QNAME  TO MQOD-OBJECTNAME.
*   MOVE W02-NAME-PREFIX  TO MQOD-DYNAMICQNAME.
*
*   COMPUTE W03-OPTIONS = MQOD-INPUT-EXCLUSIVE.
*
*   CALL 'MQOPEN' USING W03-HCONN
*                       MQOD
*                       W03-OPTIONS
*                       W03-HOBJ-MODEL
*                       W03-COMPCODE
*                       W03-REASON.
*
*   IF W03-COMPCODE NOT = MQCC-OK
*       MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
*       MOVE W03-COMPCODE  TO M01-MSG4-COMPCODE
*       MOVE W03-REASON    TO M01-MSG4-REASON
*       MOVE M01-MESSAGE-4 TO M00-MESSAGE
*   ELSE
*       MOVE MQOD-OBJECTNAME TO W02-TEMPORARY-Q
*   END-IF.
*
*   OPEN-TEMP-RESPONSE-QUEUE-EXIT.
*
*   Return to performing section.
*
*   EXIT.
*   EJECT
*

```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open an existing queue.

This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
* -----*
*   WORKING-STORAGE SECTION.
* -----*
*
*   W01 - Fields derived from the command area input
*
*   01 W01-OBJECT          PIC X(48).
*
*   W02 - MQM API fields
*
*   01 W02-HCONN          PIC S9(9) BINARY VALUE ZERO.
*   01 W02-OPTIONS        PIC S9(9) BINARY.
*   01 W02-HOBJ           PIC S9(9) BINARY.
*   01 W02-COMPCODE       PIC S9(9) BINARY.
*   01 W02-REASON         PIC S9(9) BINARY.
*
*   CMQODV defines the object descriptor (MQOD)
*

```

```

01 MQM-OBJECT-DESCRIPTOR.
   COPY CMQODV.
*
* CMQV contains constants (for setting or testing
* field values) and return codes (for testing the
* result of a call)
*
01 MQM-CONSTANTS.
   COPY CMQV SUPPRESS.
* -----*
E-OPEN-QUEUE SECTION.
* -----*
*
* This section opens the queue
*
* Initialize the Object Descriptor (MQOD) control
* block
* (The copy file initializes the remaining fields.)
*
   MOVE MQOT-Q          TO MQOD-OBJECTTYPE.
   MOVE W01-OBJECT      TO MQOD-OBJECTNAME.
*
* Initialize W02-OPTIONS to open the queue for both
* inquiring about and setting attributes
*
   COMPUTE W02-OPTIONS = MQ00-INQUIRE + MQ00-SET.
*
*
* Open the queue
*
   CALL 'MQOPEN' USING W02-HCONN
                      MQOD
                      W02-OPTIONS
                      W02-HOBJ
                      W02-COMPCODE
                      W02-REASON.
*
* Test the output from the open
*
* If the completion code is not OK, display a
* separate error message for each of the following
* errors:
*
* Q-MGR-NOT-AVAILABLE - MQM is not available
* CONNECTION-BROKEN  - MQM is no longer connected to CICS
* UNKNOWN-OBJECT-NAME - The queue does not exist
* NOT-AUTHORIZED     - The user is not authorized to open
*                     the queue
*
* For any other error, display an error message
* showing the completion and reason codes
*
   IF W02-COMPCODE NOT = MQCC-OK
      EVALUATE TRUE
*
      WHEN W02-REASON = MQRC-Q-MGR-NOT-AVAILABLE
         MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
      WHEN W02-REASON = MQRC-CONNECTION-BROKEN
         MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
      WHEN W02-REASON = MQRC-UNKNOWN-OBJECT-NAME
         MOVE M01-MESSAGE-2 TO M00-MESSAGE
*
      WHEN W02-REASON = MQRC-NOT-AUTHORIZED
         MOVE M01-MESSAGE-3 TO M00-MESSAGE
*
      WHEN OTHER
         MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
         MOVE W02-COMPCODE  TO M01-MSG4-COMPCODE
         MOVE W02-REASON    TO M01-MSG4-REASON
         MOVE M01-MESSAGE-4 TO M00-MESSAGE
      END-EVALUATE
   END-IF.
E-EXIT.
*
* Return to performing section
*
   EXIT.
EJECT

```

Closing a queue

This example demonstrates how to use the MQCLOSE call.

The variables used in this code extract are those that were set in [“Connecting to a queue manager”](#) on page 23. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
:
*
*   Close the queue
*
*   MOVE MQCO-NONE TO W03-OPTIONS.
*
*   CALL 'MQCLOSE' USING W03-HCONN
*                       W03-HOBJ
*                       W03-OPTIONS
*                       W03-COMPCODE
*                       W03-REASON.
*
*   Test the output of the MQCLOSE call. If the call
*   fails, print an error message showing the
*   completion code and reason code.
*
*   IF (W03-COMPCODE NOT = MQCC-OK) THEN
*       MOVE 'CLOSE'          TO W04-MSG4-TYPE
*       MOVE W03-COMPCODE     TO W04-MSG4-COMPCODE
*       MOVE W03-REASON       TO W04-MSG4-REASON
*       MOVE W04-MESSAGE-4 TO W00-PRINT-DATA
*       PERFORM PRINT-LINE
*       MOVE W06-CSQ4-ERROR TO W00-RETURN-CODE
*   END-IF.
*
```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call using context.

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```
:
* -----*
*   WORKING-STORAGE SECTION.
* -----*
*
*   W02 - Queues processed in this program
*
*   01 W02-TEMPORARY-Q          PIC X(48).
*
*   W03 - MQM API fields
*
*   01 W03-HCONN                PIC S9(9) BINARY VALUE ZERO.
*   01 W03-HOBJ-INQUIRY        PIC S9(9) BINARY.
*   01 W03-OPTIONS              PIC S9(9) BINARY.
*   01 W03-BUFFLEN              PIC S9(9) BINARY.
*   01 W03-COMPCODE             PIC S9(9) BINARY.
*   01 W03-REASON               PIC S9(9) BINARY.
*
*   01 W03-PUT-BUFFER.
*
*   05 W03-CSQ4BIIM.
*   COPY CSQ4VB1.
*
*   API control blocks
*
*   01 MQM-MESSAGE-DESCRIPTOR.
*   COPY CMQMDV.
*   01 MQM-PUT-MESSAGE-OPTIONS.
*   COPY CMQPMOV.
*
*   MQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
*   01 MQM-CONSTANTS.
```

```

COPY CMQV SUPPRESS.
* -----*
PROCEDURE DIVISION.
* -----*
:
*   Open queue and build message.
:
*
* Set the message descriptor and put-message options to
* the values required to create the message.
* Set the length of the message.
*
MOVE MQMT-REQUEST      TO MQMD-MSGTYPE.
MOVE MQCI-NONE          TO MQMD-CORRELID.
MOVE MQMI-NONE          TO MQMD-MSGID.
MOVE W02-TEMPORARY-Q    TO MQMD-REPLYTOQ.
MOVE SPACES             TO MQMD-REPLYTOQMGR.
MOVE 5                  TO MQMD-PRIORITY.
MOVE MQPER-NOT-PERSISTENT TO MQMD-PERSISTENCE.
COMPUTE MQPMO-OPTIONS   = MQPMO-NO-SYNCPPOINT +
                        MQPMO-DEFAULT-CONTEXT.
MOVE LENGTH OF CSQ4BIIM-MSG TO W03-BUFFLEN.
*
CALL 'MQPUT' USING W03-HCONN
                  W03-HOBJ-INQUIRY
                  MQMD
                  MQPMO
                  W03-BUFFLEN
                  W03-PUT-BUFFER
                  W03-COMPCODE
                  W03-REASON.
IF W03-COMPCODE NOT = MQCC-OK
:
END-IF.

```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call.

This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W03 - MQM API fields
*
01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
01 W03-OPTIONS        PIC S9(9) BINARY.
01 W03-COMPCODE       PIC S9(9) BINARY.
01 W03-REASON         PIC S9(9) BINARY.
01 W03-BUFFLEN        PIC S9(9) BINARY.
*
01 W03-PUT-BUFFER.
   05 W03-CSQ4BQRM.
   COPY CSQ4VB4.
*
*   API control blocks
*
01 MQM-OBJECT-DESCRIPTOR.
   COPY CMQODV.
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
01 MQM-PUT-MESSAGE-OPTIONS.
   COPY CMQPMOV.
*
* CMQV contains constants (for filling in the
* control blocks) and return codes (for testing
* the result of a call).
*
01 MQM-MQV.
COPY CMQV SUPPRESS.

```

```

* -----*
PROCEDURE DIVISION.
* -----*
:
*   Get the request message.
:
* -----*
PROCESS-QUERY SECTION.
* -----*
:
*   Build the reply message.
:
*
* Set the object descriptor, message descriptor and
* put-message options to the values required to create
* the message.
* Set the length of the message.
*
  MOVE MQMD-REPLYTOQ      TO MQOD-OBJECTNAME.
  MOVE MQMD-REPLYTOQMGR   TO MQOD-OBJECTQMGRNAME.
  MOVE MQMT-REPLY         TO MQMD-MSGTYPE.
  MOVE SPACES             TO MQMD-REPLYTOQ.
  MOVE SPACES             TO MQMD-REPLYTOQMGR.
  MOVE LOW-VALUES         TO MQMD-MSGID.
  COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
                        MQPMO-PASS-IDENTITY-CONTEXT.
  MOVE W03-HOBJ-CHECKQ    TO MQPMO-CONTEXT.
  MOVE LENGTH OF CSQ4BQRM-MSG TO W03-BUFFLEN.
*
  CALL 'MQPUT1' USING W03-HCONN
                    MQOD
                    MQMD
                    MQPMO
                    W03-BUFFLEN
                    W03-PUT-BUFFER
                    W03-COMPCODE
                    W03-REASON.
  IF W03-COMPCODE NOT = MQCC-OK
    MOVE 'MQPUT1'      TO M02-OPERATION
    MOVE MQOD-OBJECTNAME TO M02-OBJECTNAME
    PERFORM RECORD-CALL-ERROR
    PERFORM FORWARD-MSG-TO-DLQ
  END-IF.
*

```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue.

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W03 - MQM API fields
*
01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
01 W03-HOBJ-RESPONSE  PIC S9(9) BINARY.
01 W03-OPTIONS        PIC S9(9) BINARY.
01 W03-BUFFLEN        PIC S9(9) BINARY.
01 W03-DATALEN        PIC S9(9) BINARY.
01 W03-COMPCODE       PIC S9(9) BINARY.
01 W03-REASON         PIC S9(9) BINARY.
*
01 W03-GET-BUFFER.
   05 W03-CSQ4BAM.
   COPY CSQ4VB2.
*
*   API control blocks
*
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
01 MQM-GET-MESSAGE-OPTIONS.
   COPY CMQGMV.
*

```

```

*      MQV contains constants (for filling in the
*      control blocks) and return codes (for testing
*      the result of a call).
*
*      01  MQM-CONSTANTS.
*          COPY CMQV SUPPRESS.
* -----*
*      A-MAIN SECTION.
* -----*
*      :
*      :      Open response queue.
*      :
* -----*
*      PROCESS-RESPONSE-SCREEN SECTION.
* -----*
*      :
*      :      This section gets a message from the response queue.
*      :
*      :      When a correct response is received, it is
*      :      transferred to the map for display; otherwise
*      :      an error message is built.
*      :
* -----*

```

```

*
*      Set get-message options
*
*      COMPUTE MQGMO-OPTIONS = MQGMO-SYNCPPOINT +
*                               MQGMO-ACCEPT-TRUNCATED-MSG +
*                               MQGMO-NO-WAIT.
*
*      Set msgid and correlid in MQMD to nulls so that any
*      message will qualify.
*      Set length to available buffer length.
*
*      MOVE MQMI-NONE TO MQMD-MSGID.
*      MOVE MQCI-NONE TO MQMD-CORRELID.
*      MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
*      CALL 'MQGET' USING W03-HCONN
*                          W03-HOBJ-RESPONSE
*                          MQMD
*                          MQGMO
*                          W03-BUFFLEN
*                          W03-GET-BUFFER
*                          W03-DATALEN
*                          W03-COMPCODE
*                          W03-REASON.
*
*      EVALUATE TRUE
*          WHEN W03-COMPCODE NOT = MQCC-FAILED
*          :
*          :      Process the message
*          :
*          WHEN (W03-COMPCODE = MQCC-FAILED AND
*                W03-REASON = MQRC-NO-MSG-AVAILABLE)
*              MOVE M01-MESSAGE-9 TO M00-MESSAGE
*              PERFORM CLEAR-RESPONSE-SCREEN
*
*          WHEN OTHER
*              MOVE 'MQGET '      TO M01-MSG4-OPERATION
*              MOVE W03-COMPCODE TO M01-MSG4-COMPCODE
*              MOVE W03-REASON   TO M01-MSG4-REASON
*              MOVE M01-MESSAGE-4 TO M00-MESSAGE
*              PERFORM CLEAR-RESPONSE-SCREEN
*      END-EVALUATE.

```

Getting a message using the wait option

This example demonstrates how to use the MQGET call with the wait option and accepting truncated messages.

This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

*
* -----*
*      WORKING-STORAGE SECTION.

```

```

* -----*
*
*   W00 - General work fields
*
* 01 W00-WAIT-INTERVAL   PIC S9(09) BINARY VALUE 30000.
*
*   W03 - MQM API fields
*
* 01 W03-HCONN           PIC S9(9) BINARY VALUE ZERO.
* 01 W03-OPTIONS         PIC S9(9) BINARY.
* 01 W03-HOBJ-CHECKQ     PIC S9(9) BINARY.
* 01 W03-COMPCODE        PIC S9(9) BINARY.
* 01 W03-REASON          PIC S9(9) BINARY.
* 01 W03-DATALEN         PIC S9(9) BINARY.
* 01 W03-BUFFLEN         PIC S9(9) BINARY.
*
* 01 W03-MSG-BUFFER.
*   05 W03-CSQ4BCAQ.
*   COPY CSQ4VB3.
*
*   API control blocks
*
* 01 MQM-MESSAGE-DESCRIPTOR.
*   COPY CMQMDV.
* 01 MQM-GET-MESSAGE-OPTIONS.
*   COPY CMQGMV.
*
*   CMQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
* 01 MQM-MQV.
*   COPY CMQV SUPPRESS.
* -----*
*   PROCEDURE DIVISION.
* -----*
*
*   Open input queue.
*

```

```

*
*   Get and process messages.
*
*   COMPUTE MQGMO-OPTIONS = MQGMO-WAIT +
*                           MQGMO-ACCEPT-TRUNCATED-MSG +
*                           MQGMO-SYNCPPOINT.
*   MOVE LENGTH OF W03-MSG-BUFFER TO W03-BUFFLEN.
*   MOVE W00-WAIT-INTERVAL TO MQGMO-WAITINTERVAL.
*   MOVE MQMI-NONE TO MQMD-MSGID.
*   MOVE MQCI-NONE TO MQMD-CORRELID.
*
*   Make the first MQGET call outside the loop.
*
*   CALL 'MQGET' USING W03-HCONN
*                     W03-HOBJ-CHECKQ
*                     MQMD
*                     MQGMO
*                     W03-BUFFLEN
*                     W03-MSG-BUFFER
*                     W03-DATALEN
*                     W03-COMPCODE
*                     W03-REASON.
*
*   Test the output of the MQGET call using the
*   PERFORM loop that follows.
*
*   Perform whilst no failure occurs
*   - process this message
*   - reset the call parameters
*   - get another message
*   End-perform
*
*
*   Test the output of the MQGET call.  If the call
*   fails, send an error message showing the
*   completion code and reason code, unless the
*   completion code is NO-MSG-AVAILABLE.
*
*   IF (W03-COMPCODE NOT = MQCC-FAILED) OR
*      (W03-REASON NOT = MQRC-NO-MSG-AVAILABLE)

```

```

        MOVE 'MQGET '          TO M02-OPERATION
        MOVE MQ0D-OBJECTNAME    TO M02-OBJECTNAME
        PERFORM RECORD-CALL-ERROR
    END-IF.
:

```

Getting a message using signaling

This example demonstrates how to use the MQGET call with signaling. This extract is taken from the Credit Check sample application (program CSQ4CVB2) supplied with IBM MQ for z/OS.

Signaling is available only with IBM MQ for z/OS .

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W00 - General work fields
:
01 W00-WAIT-INTERVAL    PIC S9(09) BINARY VALUE 30000.
*
*   W03 - MQM API fields
*
01 W03-HCONN            PIC S9(9) BINARY VALUE ZERO.
01 W03-HOBJ-REPLYQ      PIC S9(9) BINARY.
01 W03-COMPCODE         PIC S9(9) BINARY.
01 W03-REASON           PIC S9(9) BINARY.
01 W03-DATALEN          PIC S9(9) BINARY.
01 W03-BUFFLEN          PIC S9(9) BINARY.
:
01 W03-GET-BUFFER.
05 W03-CSQ4BQRM.
COPY CSQ4VB4.
*
05 W03-CSQ4BIIM REDEFINES W03-CSQ4BQRM.
COPY CSQ4VB1.
*
05 W03-CSQ4BPGM REDEFINES W03-CSQ4BIIM.
COPY CSQ4VB5.
:
*   API control blocks
*
01 MQM-MESSAGE-DESCRIPTOR.
COPY CMQMDV.
01 MQM-GET-MESSAGE-OPTIONS.
COPY CMQGMV.
:
*   MQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
01 MQM-MQV.
COPY CMQV SUPPRESS.
* -----*
LINKAGE SECTION.
* -----*
01 L01-ECB-ADDR-LIST.
05 L01-ECB-ADDR1      POINTER.
05 L01-ECB-ADDR2      POINTER.

*
01 L02-ECBS.
05 L02-INQUIRY-ECB1    PIC S9(09) BINARY.
05 L02-REPLY-ECB2     PIC S9(09) BINARY.
01 REDEFINES L02-ECBS.
05                      PIC X(02).
05 L02-INQUIRY-ECB1-CC PIC S9(04) BINARY.
05                      PIC X(02).
05 L02-REPLY-ECB2-CC   PIC S9(04) BINARY.
*
* -----*
PROCEDURE DIVISION.
* -----*
:
* Initialize variables, open queues, set signal on
* inquiry queue.

```

```

:
* -----*
PROCESS-SIGNAL-ACCEPTED SECTION.
* -----*
* This section gets a message with signal. If a
* message is received, process it. If the signal
* is set or is already set, the program goes into
* an operating system wait.
* Otherwise an error is reported and call error set.
* -----*
*
PERFORM REPLYQ-GETSIGNAL.
*
EVALUATE TRUE
  WHEN (W03-COMPCODE = MQCC-OK AND
        W03-REASON = MQRC-NONE)
    PERFORM PROCESS-REPLYQ-MESSAGE
*
  WHEN (W03-COMPCODE = MQCC-WARNING AND
        W03-REASON = MQRC-SIGNAL-REQUEST-ACCEPTED)
    OR
    (W03-COMPCODE = MQCC-FAILED AND
     W03-REASON = MQRC-SIGNAL-OUTSTANDING)
    PERFORM EXTERNAL-WAIT
*
  WHEN OTHER
    MOVE 'MQGET SIGNAL' TO M02-OPERATION
    MOVE MQ0D-OBJECTNAME TO M02-OBJECTNAME
    PERFORM RECORD-CALL-ERROR
    MOVE W06-CALL-ERROR TO W06-CALL-STATUS
END-EVALUATE.
*
PROCESS-SIGNAL-ACCEPTED-EXIT.
* Return to performing section
EXIT.
EJECT
*

```

```

* -----*
EXTERNAL-WAIT SECTION.
* -----*
* This section performs an external CICS wait on two
* ECBs until at least one is posted. It then calls
* the sections to handle the posted ECB.
* -----*
EXEC CICS WAIT EXTERNAL
  ECBLIST(W04-ECB-ADDR-LIST-PTR)
  NUMEVENTS(2)
END-EXEC.
*
* At least one ECB must have been posted to get to this
* point. Test which ECB has been posted and perform
* the appropriate section.
*
IF L02-INQUIRY-ECB1 NOT = 0
  PERFORM TEST-INQUIRYQ-ECB
ELSE
  PERFORM TEST-REPLYQ-ECB
END-IF.
*
EXTERNAL-WAIT-EXIT.
*
Return to performing section.
*
EXIT.
EJECT
:
* -----*
REPLYQ-GETSIGNAL SECTION.
* -----*
* This section performs an MQGET call (in syncpoint with
* signal) on the reply queue. The signal field in the
* MQGMO is set to the address of the ECB.
* Response handling is done by the performing section.
* -----*
*
  COMPUTE MQGMO-OPTIONS = MQGMO-SYNCPPOINT +
                        MQGMO-SET-SIGNAL.
  MOVE W00-WAIT-INTERVAL TO MQGMO-WAITINTERVAL.

```

```

        MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
        MOVE ZEROS                      TO L02-REPLY-ECB2.
        SET MQGMO-SIGNAL1 TO ADDRESS OF L02-REPLY-ECB2.

*
*   Set msgid and correlid to nulls so that any message
*   will qualify.
*
        MOVE MQMI-NONE TO MQMD-MSGID.
        MOVE MQCI-NONE TO MQMD-CORRELID.
*
        CALL 'MQGET' USING W03-HCONN
                           W03-HOBJ-REPLYQ
                           MQMD
                           MQGMO
                           W03-BUFFLEN
                           W03-GET-BUFFER
                           W03-DATALEN
                           W03-COMPCODE
                           W03-REASON.
*
        REPLYQ-GETSIGNAL-EXIT.
*
        Return to performing section.
*
        EXIT.
        EJECT
*
        :

```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue.

This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#).

```

:
* -----*
*   WORKING-STORAGE SECTION.
* -----*
*
*   W02 - MQM API fields
*
01  W02-SELECTORCOUNT    PIC S9(9) BINARY VALUE 2.
01  W02-INTATTRCOUNT    PIC S9(9) BINARY VALUE 2.
01  W02-CHARATTRLENGTH   PIC S9(9) BINARY VALUE ZERO.
01  W02-CHARATTRS        PIC X      VALUE LOW-VALUES.
01  W02-HCONN             PIC S9(9) BINARY VALUE ZERO.
01  W02-HOBJ              PIC S9(9) BINARY.
01  W02-COMPCODE          PIC S9(9) BINARY.
01  W02-REASON            PIC S9(9) BINARY.
01  W02-SELECTORS-TABLE.
05  W02-SELECTORS        PIC S9(9) BINARY OCCURS 2 TIMES
01  W02-INTATTRS-TABLE.
05  W02-INTATTRS        PIC S9(9) BINARY OCCURS 2 TIMES
*
*   CMQODV defines the object descriptor (MQOD).
*
01  MQM-OBJECT-DESCRIPTOR.
    COPY CMQODV.
*
*   CMQV contains constants (for setting or testing field
*   values) and return codes (for testing the result of a
*   call).
*
01  MQM-CONSTANTS.
    COPY CMQV SUPPRESS.
* -----*
*   PROCEDURE DIVISION.
* -----*
*
*   Get the queue name and open the queue.
*
:
*

```

```

* Initialize the variables for the inquiry call:
*   - Set W02-SELECTORS-TABLE to the attributes whose
*     status is required
*   - All other variables are already set
*
MOVE MQIA-INHIBIT-GET TO W02-SELECTORS(1).
MOVE MQIA-INHIBIT-PUT TO W02-SELECTORS(2).

```

```

*
* Inquire about the attributes.
*
CALL 'MQINQ' USING W02-HCONN,
                  W02-HOBJ,
                  W02-SELECTORCOUNT,
                  W02-SELECTORS-TABLE,
                  W02-INTATTRCOUNT,
                  W02-INTATTRS-TABLE,
                  W02-CHARATTRLENGTH,
                  W02-CHARATTRS,
                  W02-COMPCODE,
                  W02-REASON.
*
* Test the output from the inquiry:
*
* - If the completion code is not OK, display an error
*   message showing the completion and reason codes
*
* - Otherwise, move the correct attribute status into
*   the relevant screen map fields
*
IF W02-COMPCODE NOT = MQCC-OK
  MOVE 'MQINQ'      TO M01-MSG4-OPERATION
  MOVE W02-COMPCODE TO M01-MSG4-COMPCODE
  MOVE W02-REASON   TO M01-MSG4-REASON
  MOVE M01-MESSAGE-4 TO M00-MESSAGE
*
ELSE
*   Process the changes.
*   :
*   END-IF.
*   :

```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue.

This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with IBM MQ for z/OS. For the names and locations of the sample applications on other platforms, see [Sample procedural programs \(platforms except z/OS\)](#)

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
* W02 - MQM API fields
*
01 W02-SELECTORCOUNT PIC S9(9) BINARY VALUE 2.
01 W02-INTATTRCOUNT PIC S9(9) BINARY VALUE 2.
01 W02-CHARATTRLENGTH PIC S9(9) BINARY VALUE ZERO.
01 W02-CHARATTRS PIC X VALUE LOW-VALUES.
01 W02-HCONN PIC S9(9) BINARY VALUE ZERO.
01 W02-HOBJ PIC S9(9) BINARY.
01 W02-COMPCODE PIC S9(9) BINARY.
01 W02-REASON PIC S9(9) BINARY.
01 W02-SELECTORS-TABLE.
05 W02-SELECTORS PIC S9(9) BINARY OCCURS 2 TIMES.
01 W02-INTATTRS-TABLE.
05 W02-INTATTRS PIC S9(9) BINARY OCCURS 2 TIMES.
*
* CMQODV defines the object descriptor (MQOD).
*
01 MQM-OBJECT-DESCRIPTOR.
COPY CMQODV.
*
* CMQV contains constants (for setting or testing
* field values) and return codes (for testing the
* result of a call).

```

```

*
* 01 MQM-CONSTANTS.
* COPY CMQV SUPPRESS.
* -----*
* PROCEDURE DIVISION.
* -----*

```

```

*
*   Get the queue name and open the queue.
*
*
*
*
* Initialize the variables required for the set call:
* - Set W02-SELECTORS-TABLE to the attributes to be set
* - Set W02-INTATTRS-TABLE to the required status
* - All other variables are already set
*
*   MOVE MQIA-INHIBIT-GET    TO W02-SELECTORS(1).
*   MOVE MQIA-INHIBIT-PUT    TO W02-SELECTORS(2).
*   MOVE MQQA-GET-INHIBITED TO W02-INTATTRS(1).
*   MOVE MQQA-PUT-INHIBITED TO W02-INTATTRS(2).
*
*   Set the attributes.
*
*   CALL 'MQSET' USING W02-HCONN,
*                       W02-HOBJ,
*                       W02-SELETCOUNT,
*                       W02-SELECTORS-TABLE,
*                       W02-INTATTRCOUNT,
*                       W02-INTATTRS-TABLE,
*                       W02-CHARATTRLENGTH,
*                       W02-CHARATTRS,
*                       W02-COMPCODE,
*                       W02-REASON.
*
* Test the output from the call:
*
*
* - If the completion code is not OK, display an error
*   message showing the completion and reason codes
*
*
* - Otherwise, move 'INHIBITED' into the relevant
*   screen map fields
*
*   IF W02-COMPCODE NOT = MQCC-OK
*       MOVE 'MQSET'          TO M01-MSG4-OPERATION
*       MOVE W02-COMPCODE     TO M01-MSG4-COMPCODE
*       MOVE W02-REASON       TO M01-MSG4-REASON
*       MOVE M01-MESSAGE-4 TO M00-MESSAGE
*   ELSE
*
*       Process the changes.
*
*
*   END-IF.

```

System/390 assembler-language examples

This collection of topics is mostly taken from the IBM MQ for z/OS sample applications.

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in z/OS batch.

This extract is taken from the Browse sample program (CSQ4BAA1) supplied with IBM MQ for z/OS.

```

:
WORKAREA DSECT
*
PARMLIST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
COMPCODE DS    F           Completion code
REASON   DS    F           Reason code
HCONN    DS    F           Connection handle
          ORG
PARMADDR DS    F           Address of parm field
PARMLEN  DS    H           Length of parm field

```

```

*
MQMNAME DS CL48 Queue manager name
*
*
*****
* SECTION NAME : MAINPARM *
*****
MAINPARM DS 0H
MVI MQMNAME,X'40'
MVC MQMNAME+1(L'MQMNAME-1),MQMNAME
*
* Space out first byte and initialize
*
*
* Code to address and verify parameters passed omitted
*
*
PARM1MVE DS 0H
SR R1,R3 Length of data
LA R4,MQMNAME Address for target
BCTR R1,R0 Reduce for execute
EX R1,MOVEPARM Move the data
*
*****
* EXECUTES *
*****
MOVEPARM MVC 0(*-*,R4),0(R3)
*
EJECT

```

```

*****
* SECTION NAME : MAINCONN *
*****
*
*
MAINCONN DS 0H
XC HCONN,HCONN Null connection handle
*
CALL MQCONN, X
(MQMNAME, X
HCONN, X
COMPCODE, X
REASON), X
MF=(E,PARMLIST),VL
*
LA R0,MQCC_OK Expected compcode
C R0,COMPCODE As expected?
BER R6 Yes .. return to caller
*
MVC INF4_TYP,=CL10'CONNECT '
BAL R7,ERRCODE Translate error
LA R0,8 Set exit code
ST R0,EXITCODE to 8
B ENDPROG End the program
*

```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in z/OS batch.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
* ISSUE MQI DISC REQUEST USING REENTRANT FORM
* OF CALL MACRO
*
* HCONN WAS SET BY A PREVIOUS MQCONN REQUEST
* R5 = WORK REGISTER
*
DISC DS 0H
CALL MQDISC, X
(HCONN, X
COMPCODE, X
REASON), X
VL,MF=(E,CALLLST)
*
LA R5,MQCC_OK

```

```

      C      R5,COMPCODE
      BNE    BADCALL
      :
      :
BADCALL DS    0H
:
*
*          CONSTANTS
*
      CMQA
*
*          WORKING STORAGE (RE-ENTRANT)
*
WEG3     DSECT
*
CALLLST  CALL  ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
HCONN    DS    F
COMPCODE DS    F
REASON   DS    F
*
*
LEG3     EQU    *-WKEG3
END

```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
*      R5 = WORK REGISTER.
*
OPEN     DS    0H
*
      MVC    WOD_AREA,MQOD_AREA INITIALIZE WORKING VERSION OF
*                               MQOD WITH DEFAULTS
      MVC    WOD_OBJECTNAME,MOD_Q   COPY IN THE MODEL Q NAME
      MVC    WOD_DYNAMICQNAME,DYN_Q COPY IN THE DYNAMIC Q NAME
      L      R5,=AL4(MQOO_OUTPUT)   OPEN FOR OUTPUT AND
      A      R5,=AL4(MQOO_INQUIRE) INQUIRE
      ST     R5,OPTIONS
*
*      *
*      * ISSUE MQI OPEN REQUEST USING REENTRANT
*      * FORM OF CALL MACRO
*
      CALL  MQOPEN,                      X
            (HCONN,                      X
              WOD,                        X
              OPTIONS,                   X
              HOBJ,                      X
              COMPCODE,                  X
              REASON),VL,MF=(E,CALLLST)
*
      LA    R5,MQCC_OK                  CHECK THE COMPLETION CODE
      C     R5,COMPCODE                  FROM THE REQUEST AND BRANCH
      BNE   BADCALL                     TO ERROR ROUTINE IF NOT MQCC_OK
*
      MVC    TEMP_Q,WOD_OBJECTNAME      SAVE NAME OF TEMPORARY Q
*                               CREATED BY OPEN OF MODEL Q
*
:
:
BADCALL  DS    0H
:
*
*
*      CONSTANTS:
*
MOD_Q    DC    CL48'QUERY.REPLY.MODEL'  MODEL QUEUE NAME
DYN_Q    DC    CL48'QUERY.TEMPQ.*'      DYNAMIC QUEUE NAME
*
      CMQODA DSECT=NO,LIST=YES          CONSTANT VERSION OF MQOD
      CMQA                                MQI VALUE EQUATES
*
*      WORKING STORAGE
*

```

```

DFHEISTG
HCONN DS F CONNECTION HANDLE
OPTIONS DS F OPEN OPTIONS
HOBJ DS F OBJECT HANDLE
COMPCODE DS F MQI COMPLETION CODE
REASON DS F MQI REASON CODE
TEMP_Q DS CL(MQ_Q_NAME_LENGTH) SAVED QNAME AFTER OPEN
*
WOD CMQODA DSECT=NO,LIST=YES WORKING VERSION OF MQOD
*
CALLLST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L LIST FORM
OF CALL
* MACRO
:
END

```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open a queue that has already been defined.

It shows how to specify two options. This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
* R5 = WORK REGISTER.
*
OPEN DS 0H
*
MVC WOD_AREA,MQOD_AREA INITIALIZE WORKING VERSION OF
* MQOD WITH DEFAULTS
MVC WOD_OBJECTNAME,Q_NAME SPECIFY Q NAME TO OPEN
LA R5,MQOO_INPUT_EXCLUSIVE OPEN FOR MQGET CALLS
*
ST R5,OPTIONS
*
* ISSUE MQI OPEN REQUEST USING REENTRANT FORM
* OF CALL MACRO
*
CALL MQOPEN, X
(HCONN, X
WOD, X
OPTIONS, X
HOBJ, X
COMPCODE, X
REASON),VL,MF=(E,CALLLST)
*
LA R5,MQCC_OK CHECK THE COMPLETION CODE
C R5,COMPCODE FROM THE REQUEST AND BRANCH
BNE BADCALL TO ERROR ROUTINE IF NOT MQCC_OK
*
:
BADCALL DS 0H
:
*
*
* CONSTANTS:
*
Q_NAME DC CL48'REQUEST.QUEUE' NAME OF QUEUE TO OPEN
*
CMQODA DSECT=NO,LIST=YES CONSTANT VERSION OF MQOD
CMQA MQI VALUE EQUATES
*
* WORKING STORAGE
*
DFHEISTG
HCONN DS F CONNECTION HANDLE
OPTIONS DS F OPEN OPTIONS
HOBJ DS F OBJECT HANDLE
COMPCODE DS F MQI COMPLETION CODE
REASON DS F MQI REASON CODE
*
WOD CMQODA DSECT=NO,LIST=YES WORKING VERSION OF MQOD
*
CALLLST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L LIST FORM
OF CALL
* MACRO
:
END

```

Closing a queue

This example demonstrates how to use the MQCLOSE call to close a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
* ISSUE MQI CLOSE REQUEST USING REENTRANT FROM OF
* CALL MACRO
*
*      HCONN WAS SET BY A PREVIOUS MQCONN REQUEST
*      HOBJ  WAS SET BY A PREVIOUS MQOPEN REQUEST
*      R5 = WORK REGISTER
*
CLOSE   DS    0H
        LA    R5,MQCO_NONE          NO SPECIAL CLOSE OPTIONS
        ST    R5,OPTIONS            ARE REQUIRED.
*
        CALL  MQCLOSE,              X
        (HCONN,                     X
        HOBJ,                       X
        OPTIONS,                   X
        COMPCODE,                  X
        REASON),                   X
        VL,MF=(E,CALLLST)
*
        LA    R5,MQCC_OK
        C     R5,COMPCODE
        BNE   BADCALL
*
        :
BADCALL DS    0H
        :
*
*      CONSTANTS
*
*      CMQA
*
*      WORKING STORAGE (REENTRANT)
*
WEG4    DSECT
*
CALLLST CALL , (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
HCONN   DS    F
HOBJ    DS    F
OPTIONS DS    F
COMPCODE DS    F
REASON  DS    F
*
*
LEG4     EQU   *-WKEG4
END
```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call to put a message on a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*      CONNECT TO QUEUE MANAGER
*
CONN    DS    0H
:
*
*      OPEN A QUEUE
*
OPEN     DS    0H
:
*
*      R4,R5,R6,R7 = WORK REGISTER.
*
PUT      DS    0H
        LA    R4,MQMD
        LA    R5,MQMD_LENGTH      SET UP ADDRESSES AND
        LA    R6,WMD              LENGTH FOR USE BY MVCL
        LA    R7,WMD_LENGTH      INSTRUCTION, AS MQMD IS
        MVCL  R6,R4              OVER 256 BYES LONG.
*                               INITIALIZE WORKING VERSION
*                               OF MESSAGE DESCRIPTOR
```

```

*
*   MVC   WPMO_AREA,MQPMO_AREA   INITIALIZE WORKING MQPMO
*
*   LA    R5,BUFFER_LEN         RETRIEVE THE BUFFER LENGTH
*   ST    R5,BUFFLEN            AND SAVE IT FOR MQM USE
*
*   MVC   BUFFER,TEST_MSG       SET THE MESSAGE TO BE PUT
*
*   ISSUE MQI PUT REQUEST USING REENTRANT FORM
*   OF CALL MACRO
*
*   HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*   HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
*   CALL MQPUT,                  X
*   (HCONN,                     X
*   HOBJ,                       X
*   WMD,                        X
*   WPMO,                      X
*   BUFFLEN,                   X
*   BUFFER,                    X
*   COMPCODE,                  X
*   REASON),VL,MF=(E,CALLLST)
*
*   LA    R5,MQCC_OK
*   C     R5,COMPCODE
*   BNE   BADCALL
*
*   :
BADCALL DS   0H
*   :

```

```

*
*   CONSTANTS
*
*   CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
*   CMQPMOA DSECT=NO,LIST=YES
*   CMQA
*   TEST_MSG DC CL80'THIS IS A TEST MESSAGE'
*
*   WORKING STORAGE DSECT
*
*   WORKSTG DSECT
*
*   COMPCODE DS F
*   REASON   DS F
*   BUFFLEN  DS F
*   OPTIONS  DS F
*   HCONN    DS F
*   HOBJ     DS F
*
*   BUFFER   DS CL80
*   BUFFER_LEN EQU *-BUFFER
*
*   WMD      CMQMDA DSECT=NO,LIST=NO
*   WPMO     CMQPMOA DSECT=NO,LIST=NO
*
*   CALLLST CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
*   :
*   :
*   END

```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call to open a queue, put a single message on the queue, then close the queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

*
*   :
*   *   CONNECT TO QUEUE MANAGER
*   *
*   CONN   DS   0H
*   :
*
*   R4,R5,R6,R7 = WORK REGISTER.
*
*

```

```

PUT      DS  0H
*
*      MVC  WOD_AREA,MQOD_AREA      INITIALIZE WORKING VERSION OF
*                                     MQOD WITH DEFAULTS
*      MVC  WOD_OBJECTNAME,Q_NAME  SPECIFY Q NAME FOR PUT1
*
*      LA   R4,MQMD                SET UP ADDRESSES AND
*      LA   R5,MQMD_LENGTH          LENGTH FOR USE BY MVCL
*      LA   R6,WMD                  INSTRUCTION, AS MQMD IS
*      LA   R7,WMD_LENGTH           OVER 256 BYES LONG.
*      MVCL R6,R4                  INITIALIZE WORKING VERSION
*                                     OF MESSAGE DESCRIPTOR

```

```

*
*      MVC  WPMO_AREA,MQPMO_AREA    INITIALIZE WORKING MQPMO
*
*      LA   R5,BUFFER_LEN           RETRIEVE THE BUFFER LENGTH
*      ST   R5,BUFFLEN              AND SAVE IT FOR MQM USE
*
*      MVC  BUFFER,TEST_MSG         SET THE MESSAGE TO BE PUT
*
*      * ISSUE MQI PUT REQUEST USING REENTRANT FORM OF CALL MACRO
*
*      HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*      HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
*      CALL MQPUT1,                  X
*          (HCONN,                   X
*           LMQOD,                   X
*           LMQMD,                   X
*           LMQPMO,                  X
*           BUFFERLENGTH,            X
*           BUFFER,                  X
*           COMPCODE,                X
*           REASON),VL,MF=(E,CALLST)
*
*      LA   R5,MQCC_OK
*      C    R5,COMPCODE
*      BNE  BADCALL
*
*      :
BADCALL  DS  0H
*
*

```

```

*      CONSTANTS
*
*      CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
*      CMQPMOA DSECT=NO,LIST=YES
*      CMQODA DSECT=NO,LIST=YES
*      CMQA
*
*      TEST_MSG DC CL80'THIS IS ANOTHER TEST MESSAGE'
*      Q_NAME   DC CL48'TEST.QUEUE.NAME'
*
*      WORKING STORAGE DSECT
*
*      WORKSTG  DSECT
*
*      COMPCODE DS F
*      REASON   DS F
*      BUFFLEN  DS F
*      OPTIONS  DS F
*      HCONN    DS F
*      HOBJ     DS F
*
*      BUFFER   DS CL80
*      BUFFER_LEN EQU *-BUFFER
*
*      WOD      CMQODA DSECT=NO,LIST=YES    WORKING VERSION OF MQOD
*      WMD      CMQMDA DSECT=NO,LIST=NO
*      WPMO     CMQPMOA DSECT=NO,LIST=NO
*
*      CALLLST  CALL  ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
*      :
*      END

```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
*   CONNECT TO QUEUE MANAGER
*
CONN    DS  0H
:
*
*   OPEN A QUEUE FOR GET
*
OPEN     DS  0H
:
*
*   R4,R5,R6,R7 = WORK REGISTER.
*
GET  DS  0H
    LA  R4,MQMD                SET UP ADDRESSES AND
    LA  R5,MQMD_LENGTH          LENGTH FOR USE BY MVCL
    LA  R6,WMD                  INSTRUCTION, AS MQMD IS
    LA  R7,WMD_LENGTH           OVER 256 BYES LONG.
    MVCL R6,R4                  INITIALIZE WORKING VERSION
                                OF MESSAGE DESCRIPTOR
*
*   MVC  WGM0_AREA,MQGM0_AREA   INITIALIZE WORKING MQGM0
*
    LA  R5,BUFFER_LEN           RETRIEVE THE BUFFER LENGTH
    ST  R5,BUFFLEN              AND SAVE IT FOR MQM USE
*
*
*   ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*   HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*   HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
    CALL  MQGET,                X
          (HCONN,                X
           HOBJ,                  X
           WMD,                    X
           WGM0,                  X
           BUFFLEN,               X
           BUFFER,                 X
           DATALEN,              X
           COMPCODE,              X
           REASON),              X
          VL,MF=(E,CALLLST)
*
    LA  R5,MQCC_OK
    C   R5,COMPCODE
    BNE BADCALL
*
:
BADCALL DS  0H
:

```

```

*
*   CONSTANTS
*
    CMQMDA DSECT=NO,LIST=YES
    CMQGM0A DSECT=NO,LIST=YES
    CMQA
*
*   WORKING STORAGE DSECT
*
WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80

```

```

BUFFER_LEN EQU *-BUFFER
*
WMD          CMQMDA DSECT=NO,LIST=NO
WGMO         CMQGMOA DSECT=NO,LIST=NO
*
CALLLST     CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
:
:
END

```

Getting a message using the wait option

This example demonstrates how to use the wait option of the MQGET call.

This code accepts truncated messages. This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*      CONNECT TO QUEUE MANAGER
CONN     DS  0H
:
*      OPEN A QUEUE FOR GET
OPEN     DS  0H
:
*      R4,R5,R6,R7 = WORK REGISTER.
GET      DS  0H
        LA  R4,MQMD                SET UP ADDRESSES AND
        LA  R5,MQMD_LENGTH          LENGTH FOR USE BY MVCL
        LA  R6,WMD                  INSTRUCTION, AS MQMD IS
        LA  R7,WMD_LENGTH           OVER 256 BYES LONG.
        MVCL R6,R4                  INITIALIZE WORKING VERSION
*                                  OF MESSAGE DESCRIPTOR

*
*      MVC  WGMO_AREA,MQGMO_AREA    INITIALIZE WORKING MQGMO
L        R5,=AL4(MQGMO_WAIT)
A        R5,=AL4(MQGMO_ACCEPT_TRUNCATED_MSG)
ST       R5,WGMO_OPTIONS
MVC     WGMO_WAITINTERVAL,TWO_MINUTES  WAIT UP TO TWO
                                         MINUTES BEFORE
                                         FAILING THE
                                         CALL

*
        LA  R5,BUFFER_LEN           RETRIEVE THE BUFFER LENGTH
        ST  R5,BUFFLEN              AND SAVE IT FOR MQM USE

*
*      ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*      HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*      HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
        CALL MQGET,                  X
        (HCONN,                      X
        HOBJ,                        X
        WMD,                        X
        WGMO,                      X
        BUFFLEN,                   X
        BUFFER,                    X
        DATALEN,                  X
        COMPCODE,                  X
        REASON),                   X
        VL,MF=(E,CALLLST)

*
        LA  R5,MQCC_OK              DID THE MQGET REQUEST
C        R5,COMPCODE                WORK OK?
BE       GETOK                      YES, SO GO AND PROCESS.
        LA  R5,MQCC_WARNING          NO, SO CHECK FOR A WARNING.
C        R5,COMPCODE                IS THIS A WARNING?
BE       CHECK_W                    YES, SO CHECK THE REASON.

*
        LA  R5,MQRC_NO_MSG_AVAILABLE IT MUST BE AN ERROR.
                                         IS IT DUE TO AN EMPTY
C        R5,REASON                  QUEUE?
BE       NOMSG                      YES, SO HANDLE THE ERROR
B        BADCALL                    NO, SO GO TO ERROR ROUTINE

*
CHECK_W   DS  0H
        LA  R5,MQRC_TRUNCATED_MSG_ACCEPTED  IS THIS A
                                         TRUNCATED

```

```

      C   R5,REASON          MESSAGE?
      BE  GETOK              YES, SO GO AND PROCESS.
      B   BADCALL            NO, SOME OTHER WARNING

*
NOMSG  DS  0H
      .
      .
GETOK   DS  0H
      .

```

```

BADCALL DS  0H
      .
*
*      CONSTANTS
*
      CMQMDA DSECT=NO,LIST=YES
      CMQGMOA DSECT=NO,LIST=YES
      CMQA
*
TWO_MINUTES DC F'120000'      GET WAIT INTERVAL
*
*      WORKING STORAGE DSECT

```

```

*
WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER    DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD       CMQMDA DSECT=NO,LIST=NO
WGMO      CMQGMOA DSECT=NO,LIST=NO
*
CALLLIST  CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
      .
      .
      END

```

Getting a message using signaling

This example demonstrates how to use the MQGET call to set a signal so that you are notified when a suitable message arrives on a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

:
*
*      CONNECT TO QUEUE MANAGER
*
CONN    DS  0H
      .
*
*      OPEN A QUEUE FOR GET
*
OPEN     DS  0H
      .
*
*      R4,R5,R6,R7 = WORK REGISTER.
*
GET      DS  0H
      LA  R4,MQMD              SET UP ADDRESSES AND
      LA  R5,MQMD_LENGTH      LENGTH FOR USE BY MVCL
      LA  R6,WMD              INSTRUCTION, AS MQMD IS
      LA  R7,WMD_LENGTH      OVER 256 BYES LONG.
      MVCL R6,R4              INITIALIZE WORKING VERSION
*                               OF MESSAGE DESCRIPTOR

```

```

*
MVC     WGMO_AREA,MQGMO_AREA  INITIALIZE WORKING MQGMO
LA      R5,MQGMO_SET_SIGNAL
ST      R5,WGMO_OPTIONS

```

```

MVC   WGM0_WAITINTERVAL,FIVE_MINUTES   WAIT UP TO FIVE
                                           MINUTES BEFORE
                                           FAILING THE CALL
*
*
XC    SIG_ECB,SIG_ECB   CLEAR THE ECB
LA    R5,SIG_ECB        GET THE ADDRESS OF THE ECB
ST    R5,WGM0_SIGNAL1   AND PUT IT IN THE WORKING
                                           MQGMO
*
*
LA    R5,BUFFER_LEN     RETRIEVE THE BUFFER LENGTH
ST    R5,BUFFLEN        AND SAVE IT FOR MQM USE
*
*
*   ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
HCONN WAS SET BY PREVIOUS MQCONN REQUEST
HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
CALL  MQGET,            X
      (HCONN,           X
      HOBJ,             X
      WMD,              X
      WGM0,             X
      BUFFLEN,          X
      BUFFER,           X
      DATALEN,         X
      COMPCODE,         X
      REASON),          X
      VL,MF=(E,CALLST)
*
LA    R5,MQCC_OK        DID THE MQGET REQUEST
C     R5,COMPCODE        WORK OK?
BE    GETOK             YES, SO GO AND PROCESS.
LA    R5,MQCC_WARNING    NO, SO CHECK FOR A WARNING.
C     R5,COMPCODE        IS THIS A WARNING?
BE    CHECK_W           YES, SO CHECK THE REASON.
B     BADCALL           NO, SO GO TO ERROR ROUTINE
*

```

```

CHECK_W DS 0H
LA      R5,MQRC_SIGNAL_REQUEST_ACCEPTED
C       R5,REASON        SIGNAL REQUEST SIGNAL SET?
BNE     BADCALL          NO, SOME ERROR OCCURRED
B       DOWORK           YES, SO DO SOMETHING
                                           ELSE
*
*
CHECKSIG DS 0H
CLC     SIG_ECB+1(3),=AL3(MQEC_MSG_ARRIVED)
                                           IS A MESSAGE AVAILABLE?
BE      GET              YES, SO GO AND GET IT
*
CLC     SIG_ECB+1(3),=AL3(MQEC_WAIT_INTERVAL_EXPIRED)
                                           HAVE WE WAITED LONG ENOUGH?
BE      NOMSG            YES, SO SAY NO MSG AVAILABLE
B       BADCALL          IF IT'S ANYTHING ELSE
                                           GO TO ERROR ROUTINE.
*
*
DOWORK  DS 0H
        :
        TM SIG_ECB,X'40'   HAS THE SIGNAL ECB BEEN POSTED?
        BO CHECKSIG        YES, SO GO AND CHECK WHY
        B   DOWORK         NO, SO GO AND DO MORE WORK
*
NOMSG   DS 0H
        :
GETOK   DS 0H
        :
BADCALL DS 0H
        :
*
*   CONSTANTS
*
CMQMDA DSECT=NO,LIST=YES
CMQGMOA DSECT=NO,LIST=YES
CMQA
*
FIVE_MINUTES DC F'300000'   GET SIGNAL INTERVAL
*
*   WORKING STORAGE DSECT
*

```

```

WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
SIG_ECB  DS F

```

```

*
BUFFER    DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD        CMQMDA DSECT=NO,LIST=NO
WGMO       CMQGMOA DSECT=NO,LIST=NO
*
CALLLIST  CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
:
:
END

```

Inquiring about and setting the attributes of a queue

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue and to use the MQSET call to change the attributes of a queue.

This extract is taken from the Queue Attributes sample application (program CSQ4CAC1) supplied with IBM MQ for z/OS.

```

:
DFHEISTG DSECT
:
OBJDESC  CMQODA LIST=YES   Working object descriptor
*
SELECTORCOUNT DS F      Number of selectors
INTATTRCOUNT DS F      Number of integer attributes
CHARATTRLENGTH DS F      char attributes length
CHARATTRS      DS C      Area for char attributes
*
OPTIONS DS F      Command options
HCONN   DS F      Handle of connection
HOBJ    DS F      Handle of object
COMPCODE DS F      Completion code
REASON  DS F      Reason code
SELECTOR DS 2F     Array of selectors
INTATTRS DS 2F     Array of integer attributes
:
OBJECT   DS CL(MQ_Q_NAME_LENGTH) Name of queue
:
CALLLIST CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*****
*
:
PROGRAM EXECUTION STARTS HERE
*
:
CSQ4CAC1 DFHEIENT CODEREG=(R3),DATAREG=(R13)
:
*
:
Initialize the variables for the set call
*
SR R0,R0          Clear register zero
ST R0,CHARATTRLENGTH Set char length to zero
LA R0,2           Load to set
ST R0,SELECTORCOUNT selectors add
ST R0,INTATTRCOUNT integer attributes
*
LA R0,MQIA_INHIBIT_GET Load q attribute selector
ST R0,SELECTOR+0       Place in field
LA R0,MQIA_INHIBIT_PUT Load q attribute selector
ST R0,SELECTOR+4       Place in field
*
UPDTEST DS 0H
CLC ACTION,CINHIB      Are we inhibiting?
BE UPDINHBT           Yes branch to section
*
CLC ACTION,CALLOW      Are we allowing?
BE UPDALLOW           Yes branch to section
*
MVC M00_MSG,M01_MSG1   Invalid request

```

	BR	R6	Return to caller	
*				
UPDINHBT DS 0H				
	MVC	UPDTYPE,CINHIBIT	Indicate action type	
	LA	R0,MQQA_GET_INHIBITED	Load attribute value	
	ST	R0,INTATTRS+0	Place in field	
	LA	R0,MQQA_PUT_INHIBITED	Load attribute value	
	ST	R0,INTATTRS+4	Place in field	
	B	UPDCALL	Go and do call	
*				
UPDALLOW DS 0H				
	MVC	UPDTYPE,CALLOWED	Indicate action type	
	LA	R0,MQQA_GET_ALLOWED	Load attribute value	
	ST	R0,INTATTRS+0	Place in field	
	LA	R0,MQQA_PUT_ALLOWED	Load attribute value	
	ST	R0,INTATTRS+4	Place in field	
	B	UPDCALL	Go and do call	
*				
UPDCALL DS 0H				
	CALL	MQSET,	C	
		(HCONN,	C	
		HOBJ,	C	
		SELECTORCOUNT,	C	
		SELECTOR,	C	
		INTATTRCOUNT,	C	
		INTATTRS,	C	
		CHARATTRLENGTH,	C	
		CHARATTRS,	C	
		COMPCODE,	C	
		REASON),	C	
		VL,MF=(E,CALLLIST)		
*				
	LA	R0,MQCC_OK	Load expected compcode	
	C	R0,COMPCODE	Was set successful?	
	:			
*	SECTION NAME	: INQUIRE		*
*	FUNCTION	: Inquires on the objects attributes		*
*	CALLED BY	: PROCESS		*
*	CALLS	: OPEN, CLOSE, CODES		*
*	RETURN	: To Register 6		*
INQUIRE DS 0H				
	:			
*				
Initialize the variables for the inquire call				
*				
	SR	R0,R0	Clear register zero	
	ST	R0,CHARATTRLENGTH	Set char length to zero	
	LA	R0,2	Load to set	
	ST	R0,SELECTORCOUNT	selectors add	
	ST	R0,INTATTRCOUNT	integer attributes	
*				
	LA	R0,MQIA_INHIBIT_GET	Load attribute value	
	ST	R0,SELECTOR+0	Place in field	
	LA	R0,MQIA_INHIBIT_PUT	Load attribute value	
	ST	R0,SELECTOR+4	Place in field	
	CALL	MQINQ,	C	
		(HCONN,	C	
		HOBJ,	C	
		SELECTORCOUNT,	C	
		SELECTOR,	C	
		INTATTRCOUNT,	C	
		INTATTRS,	C	
		CHARATTRLENGTH,	C	
		CHARATTRS,	C	
		COMPCODE,	C	
		REASON),	C	
		VL,MF=(E,CALLLIST)		
	LA	R0,MQCC_OK	Load expected compcode	
	C	R0,COMPCODE	Was inquire successful?	
	:			

PL/I examples

The use of PL/I is supported by z/OS only. This collection of topics demonstrates techniques using PL/I examples.

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in z/OS batch.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* STRUCTURE BASED ON PARAMETER INPUT AREA (PARAM) */
*****/
DCL 1 INPUT_PARAM      BASED(ADDR(PARAM)),
    2 PARAM_LENGTH     FIXED BIN(15),
    2 PARAM_MQMNAME    CHAR(48);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL MQMNAME            CHAR(48);
DCL COMPCODE           BINARY FIXED (31);
DCL REASON             BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
:
/*****
/* COPY QUEUE MANAGER NAME PARAMETER
/* TO LOCAL STORAGE */
*****/
MQMNAME = ' ';
MQMNAME = SUBSTR(PARAM_MQMNAME,1,PARAM_LENGTH);
:
/*****
/* CONNECT FROM THE QUEUE MANAGER */
*****/
CALL MQCONN (MQMNAME, /* MQM SYSTEM NAME */
            HCONN,    /* CONNECTION HANDLE */
            COMPCODE, /* COMPLETION CODE */
            REASON);  /* REASON CODE */

/*****
/* TEST THE COMPLETION CODE OF THE CONNECT CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
*****/
IF COMPCODE /= MQCC_OK
  THEN DO;
  :
  CALL ERROR_ROUTINE;
END;
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in z/OS batch.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE           BINARY FIXED (31);
DCL REASON             BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
:
/*****
/* DISCONNECT FROM THE QUEUE MANAGER */
*****/
CALL MQDISC (HCONN, /* CONNECTION HANDLE */
            COMPCODE, /* COMPLETION CODE */
            REASON); /* REASON CODE */

/*****
/* TEST THE COMPLETION CODE OF THE DISCONNECT CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
*****/
```

```

/*****
  IF COMPCODE = MQCC_OK
    THEN DO;
    :
    CALL ERROR_ROUTINE;
  END;
*****/

```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

      %INCLUDE SYSLIB(CMQP);
      %INCLUDE SYSLIB(CMQEPP);
      :
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
:
DCL MODEL_QUEUE_NAME  CHAR(48) INIT('PL1.REPLY.MODEL');
DCL DYNAMIC_NAME_PREFIX CHAR(48) INIT('PL1.TEMPQ.*');
DCL DYNAMIC_QUEUE_NAME CHAR(48) INIT(' ');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR */
*****/
DCL 1 LMQOD LIKE MQOD;
:
/*****
/* SET UP OBJECT DESCRIPTOR FOR OPEN OF REPLY QUEUE */
*****/
LMQOD.OBJECTTYPE =MQOT_Q;
LMQOD.OBJECTNAME = MODEL_QUEUE_NAME;
LMQOD.DYNAMICQNAME = DYNAMIC_NAME_PREFIX;
OPTIONS = MQOO_INPUT_EXCLUSIVE;

      CALL MQOPEN (HCONN,
                  LMQOD,
                  OPTIONS,
                  HOBJ,
                  COMPCODE,
                  REASON);

/*****
/* TEST THE COMPLETION CODE OF THE OPEN CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
/* IF THE CALL HAS SUCCEEDED THEN EXTRACT THE NAME OF */
/* THE NEWLY CREATED DYNAMIC QUEUE FROM THE OBJECT */
/* DESCRIPTOR. */
*****/
      IF COMPCODE = MQCC_OK
        THEN DO;
        :
        CALL ERROR_ROUTINE;
      END;
      ELSE
        DYNAMIC_QUEUE_NAME = LMQOD_OBJECTNAME;

```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open an existing queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

      %INCLUDE SYSLIB(CMQP);
      %INCLUDE SYSLIB(CMQEPP);
      :
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);

```

```

DCL HCONN          BINARY FIXED (31);
DCL HOBJ           BINARY FIXED (31);
DCL OPTIONS        BINARY FIXED (31);
:
DCL QUEUE_NAME     CHAR(48) INIT('PL1.LOCAL.QUEUE');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR
*****/
DCL 1 LMQOD LIKE MQOD;
:
/*****
/* SET UP OBJECT DESCRIPTOR FOR OPEN OF REPLY QUEUE
*****/
LMQOD.OBJECTTYPE = MQOT_Q;
LMQOD.OBJECTNAME = QUEUE_NAME;
OPTIONS = MQOO_INPUT_EXCLUSIVE;

CALL MQOPEN (HCONN,
             LMQOD,
             OPTIONS,
             HOBJ,
             COMPCODE,
             REASON);

/*****
/* TEST THE COMPLETION CODE OF THE OPEN CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.
*****/
      IF COMPCODE /= MQCC_OK
      THEN DO;
      :
      CALL ERROR_ROUTINE;
      END;

```

Closing a queue

This example demonstrates how to use the MQCLOSE call.

This extract is not taken from the sample applications supplied with IBM MQ.

```

%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS
*****/
DCL COMPCODE       BINARY FIXED (31);
DCL REASON         BINARY FIXED (31);
DCL HCONN          BINARY FIXED (31);
DCL HOBJ           BINARY FIXED (31);
DCL OPTIONS        BINARY FIXED (31);
:
/*****
/* SET CLOSE OPTIONS
*****/
OPTIONS=MQCO_NONE;

/*****
/* CLOSE QUEUE
*****/
CALL MQCLOSE (HCONN, /* CONNECTION HANDLE
                HOBJ,  /* OBJECT HANDLE
                OPTIONS, /* CLOSE OPTIONS
                COMPCODE, /* COMPLETION CODE
                REASON); /* REASON CODE

/*****
/* TEST THE COMPLETION CODE OF THE CLOSE CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.
*****/
      IF COMPCODE /= MQCC_OK
      THEN DO;
      :
      CALL ERROR_ROUTINE;
      END;

```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call using context.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
DCL BUFFLEN           BINARY FIXED (31);
DCL BUFFER            CHAR(80);
:
DCL PL1_TEST_MESSAGE  CHAR(80)
INIT('***** THIS IS A TEST MESSAGE *****');
:
/*****
/* LOCAL COPY OF MESSAGE DESCRIPTOR
/* AND PUT MESSAGE OPTIONS
*****/
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQPMO LIKE MQPMO;
:
/*****
/* SET UP MESSAGE DESCRIPTOR
*****/
LMQMD.MSGTYPE = MQMT_DATAGRAM;
LMQMD.PRIORITY = 1;
LMQMD.PERSISTENCE = MQPER_PERSISTENT;
LMQMD.REPLYTOQ = ' ';
LMQMD.REPLYTOQMGR = ' ';
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP PUT MESSAGE OPTIONS
*****/
LMQPMO.OPTIONS = MQPMO_NO_SYNCPOINT;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER AND THE MESSAGE
*****/
BUFFLEN = LENGTH(BUFFER);
BUFFER = PL1_TEST_MESSAGE;
/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
/*
/*****
CALL MQPUT (HCONN,
           HOBJ,
           LMQMD,
           LMQPMO,
           BUFFLEN,
           BUFFER,
           COMPCODE,
           REASON);

/*****
/* TEST THE COMPLETION CODE OF THE PUT CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.
*****/
IF COMPCODE /= MQCC_OK
THEN DO;
:
CALL ERROR_ROUTINE;
END;
```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQEPP);
%INCLUDE SYSLIB(CMQP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE      BINARY FIXED (31);
DCL REASON        BINARY FIXED (31);
DCL HCONN         BINARY FIXED (31);
DCL OPTIONS       BINARY FIXED (31);
DCL BUFFLEN       BINARY FIXED (31);
DCL BUFFER        CHAR(80);
:
DCL REPLY_TO_QUEUE CHAR(48) INIT('PL1.REPLY.QUEUE');
DCL QUEUE_NAME     CHAR(48) INIT('PL1.LOCAL.QUEUE');
DCL PL1_TEST_MESSAGE CHAR(80)
INIT('***** THIS IS ANOTHER TEST MESSAGE *****');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR, MESSAGE DESCRIPTOR */
/* AND PUT MESSAGE OPTIONS */
*****/
DCL 1 LMQOD LIKE MQOD;
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQPMO LIKE MQPMO;
:
/*****
/* SET UP OBJECT DESCRIPTOR AS REQUIRED. */
*****/
LMQOD.OBJECTTYPE = MQOT_Q;
LMQOD.OBJECTNAME = QUEUE_NAME;

/*****
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED. */
*****/
LMQMD.MSGTYPE = MQMT_REQUEST;
LMQMD.PRIORITY = 5;
LMQMD.PERSISTENCE = MQPER_PERSISTENT;
LMQMD.REPLYTOQ = REPLY_TO_QUEUE;
LMQMD.REPLYTOQMGR = ' ' ;
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP PUT MESSAGE OPTIONS AS REQUIRED */
*****/
LMQPMO.OPTIONS = MQPMO_NO_SYNCPOINT;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER AND THE MESSAGE */
*****/
BUFFLEN = LENGTH(BUFFER);
BUFFER = PL1_TEST_MESSAGE;

CALL MQPUT1 (HCONN,
             LMQOD,
             LMQMD,
             LMQPMO,
             BUFFLEN,
             BUFFER,
             COMPCODE,
             REASON);

/*****
/* TEST THE COMPLETION CODE OF THE PUT1 CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING */
/* THE COMPLETION CODE AND THE REASON CODE. */
*****/
IF COMPCODE /= MQCC_OK
THEN DO;
:
CALL ERROR_ROUTINE;
END;
```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON             BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL BUFFLEN           BINARY FIXED (31);
DCL DATALEN          BINARY FIXED (31);
DCL BUFFER            CHAR(80);
:

/*****
/* LOCAL COPY OF MESSAGE DESCRIPTOR AND
/* GET MESSAGE OPTIONS */
*****/
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQGMO LIKE MQGMO;
:

/*****
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED. */
/* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST */
/* AVAILABLE MESSAGE WILL BE RETRIEVED. */
*****/
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP GET MESSAGE OPTIONS AS REQUIRED. */
*****/
LMQGMO.OPTIONS = MQGMO_NO_SYNCPOINT;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER. */
*****/
BUFFLEN = LENGTH(BUFFER);

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
/*
*****/

CALL MQGET (HCONN,
            HOBJ,
            LMQMD,
            LMQGMO,
            BUFFLEN,
            BUFFER,
            DATALEN,
            COMPCODE,
            REASON);

/*****
/* TEST THE COMPLETION CODE OF THE GET CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
*****/
IF COMPCODE /= MQCC_OK
THEN DO;
:
CALL ERROR_ROUTINE;
END;
```

Getting a message using the wait option

This example demonstrates how to use the MQGET call with the wait option and accepting truncated messages.

This extract is not taken from the sample applications supplied with IBM MQ.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL BUFFLEN           BINARY FIXED (31);
DCL DATALEN          BINARY FIXED (31);
DCL BUFFER            CHAR(80);
:
/*****
/* LOCAL COPY OF MESSAGE DESCRIPTOR AND GET MESSAGE */
/* OPTIONS */
*****/
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQGMO LIKE MQGMO;
:
/*****
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED. */
/* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST */
/* AVAILABLE MESSAGE WILL BE RETRIEVED. */
*****/
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP GET MESSAGE OPTIONS AS REQUIRED. */
/* WAIT INTERVAL SET TO ONE MINUTE. */
*****/
LMQGMO.OPTIONS = MQGMO_WAIT +
                 MQGMO_ACCEPT_TRUNCATED_MSG +
                 MQGMO_NO_SYNCPOINT;
LMQGMO.WAITINTERVAL=60000;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER. */
*****/
BUFFLEN = LENGTH(BUFFER);

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST. */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST. */
/*
*****/

CALL MQGET (HCONN,
            HOBJ,
            LMQMD,
            LMQGMO,
            BUFFERLEN,
            BUFFER,
            DATALEN,
            COMPCODE,
            REASON);

/*****
/* TEST THE COMPLETION CODE OF THE GET CALL. */
/* TAKE APPROPRIATE ACTION BASED ON COMPLETION CODE AND */
/* REASON CODE. */
*****/

SELECT(COMPCODE);
  WHEN (MQCC_OK) DO;    /* GET WAS SUCCESSFUL */
  :
  END;
  WHEN (MQCC_WARNING) DO;
    IF REASON = MQRC_TRUNCATED_MSG_ACCEPTED
      THEN DO;          /* GET WAS SUCCESSFUL */
```

```

        :
        END;
        ELSE DO;
        :
        CALL ERROR_ROUTINE;
        END;
    END;
    WHEN (MQCC_FAILED) DO;
    :
    CALL ERROR_ROUTINE;
    END;
    END;
    OTHERWISE;
    END;

```

Getting a message using signaling

A code extract that demonstrates how to use the MQGET call with signaling.

Signaling is available only with IBM MQ for z/OS .

This extract is not taken from the sample applications supplied with IBM MQ.

```

        %INCLUDE SYSLIB(CMQP);
        %INCLUDE SYSLIB(CMQEPP);
        :
        /*****
        /* WORKING STORAGE DECLARATIONS
        *****/
        DCL COMPCODE          BINARY FIXED (31);
        DCL REASON            BINARY FIXED (31);
        DCL HCONN             BINARY FIXED (31);
        DCL HOBJ              BINARY FIXED (31);
        DCL DATALEN          BINARY FIXED (31);
        DCL BUFFLEN           BINARY FIXED (31);
        DCL BUFFER            CHAR(80);
        :
        DCL ECB_FIXED         FIXED BIN(31);
        DCL 1 ECB_OVERLAY BASED(ADDR(ECB_FIXED)),
            3 ECB_WAIT      BIT,
            3 ECB_POSTED    BIT,
            3 ECB_FLAG3_8   BIT(6),
            3 ECB_CODE      PIC'999';
        :
        /*****
        /* LOCAL COPY OF MESSAGE DESCRIPTOR AND GET MESSAGE
        /* OPTIONS
        *****/
        DCL 1 LMQMD LIKE MQMD;
        DCL 1 LMQGMO LIKE MQGMO;
        :
        /*****
        /* CLEAR ECB FIELD.
        *****/
        ECB_FIXED = 0;
        :
        /*****
        /* SET UP MESSAGE DESCRIPTOR AS REQUIRED.
        /* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST
        /* AVAILABLE MESSAGE WILL BE RETRIEVED.
        *****/
        LMQMD.MSGID = MQMI_NONE;
        LMQMD.CORRELID = MQCI_NONE;
        /*****
        /* SET UP GET MESSAGE OPTIONS AS REQUIRED.
        /* WAIT INTERVAL SET TO ONE MINUTE.
        *****/
        LMQGMO.OPTIONS = MQGMO_SET_SIGNAL +
                        MQGMO_NO_SYNCPOINT;
        LMQGMO.WAITINTERVAL=60000;
        LMQGMO.SIGNAL1 = ADDR(ECB_FIXED);

        /*****
        /* SET UP LENGTH OF MESSAGE BUFFER.
        /* CALL MESSGE RETRIEVAL ROUTINE.
        *****/
        BUFFLEN = LENGTH(BUFFER);
        CALL GET_MSG;

```

```

/*****
/* TEST THE COMPLETION CODE OF THE GET CALL.          */
/* TAKE APPROPRIATE ACTION BASED ON COMPLETION CODE AND */
/* REASON CODE.                                         */
*****/

SELECT;
  WHEN ((COMPCODE = MQCC_OK) &
        (REASON = MQCC_NONE)) DO
    :
    CALL MSG_ROUTINE;
    :
  END;
  WHEN ((COMPCODE = MQCC_WARNING) &
        (REASON = MQRC_SIGNAL_REQUEST_ACCEPTED)) DO;
    :
    CALL DO_WORK;
    :
  END;
  WHEN ((COMPCODE = MQCC_FAILED) &
        (REASON = MQRC_SIGNAL_OUTSTANDING)) DO;
    :
    CALL DO_WORK;
    :
  END;
  OTHERWISE DO;          /* FAILURE CASE */
/*****
/* ISSUE AN ERROR MESSAGE SHOWING THE COMPLETION CODE */
/* AND THE REASON CODE.                               */
*****/
    :
    CALL ERROR_ROUTINE;
    :
  END;
END;
:

```

```

DO_WORK: PROC;
:
  IF ECB_POSTED
  THEN DO;
    SELECT(ECB_CODE);
    WHEN(MQEC_MSG_ARRIVED) DO;
      :
      CALL GET_MSG;
      :
    END;
    WHEN(MQEC_WAIT_INTERVAL_EXPIRED) DO;
      :
      CALL NO_MSG;
      :
    END;
    OTHERWISE DO;          /* FAILURE CASE */
/*****
/* ISSUE AN ERROR MESSAGE SHOWING THE COMPLETION CODE */
/* AND THE REASON CODE.                               */
*****/
      :
      CALL ERROR_ROUTINE;
      :
    END;
  END;
END;
:
END DO_WORK;

GET_MSG: PROC;

```

```

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.          */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.           */
/* MD AND GMO SET UP AS REQUIRED.                      */
/*                                                     */
*****/

CALL MQGET (HCONN,

```

```

        HOBJ,
        LMQMD,
        LMQGMO,
        BUFLLEN,
        BUFFER,
        DATALEN,
        COMPCODE,
        REASON);

END GET_MSG;

NO_MSG: PROC;
:
END NO_MSG;

```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

        %INCLUDE SYSLIB(CMQP);
        %INCLUDE SYSLIB(CMQEPP);
        :
/*****
/* WORKING STORAGE DECLARATIONS
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
DCL SELECTORCOUNT   BINARY FIXED (31);
DCL INTATTRCOUNT    BINARY FIXED (31);
DCL 1 SELECTOR_TABLE,
    3 SELECTORS(5)    BINARY FIXED (31);
DCL 1 INTATTR_TABLE,
    3 INTATTRS(5)     BINARY FIXED (31);
DCL CHARATTRLENGTH   BINARY FIXED (31);
DCL CHARATTRS        CHAR(100);
:

/*****
/* SET VARIABLES FOR INQUIRE CALL
/* INQUIRE ON THE CURRENT QUEUE DEPTH
*****/

        SELECTORS(01) = MQIA_CURRENT_Q_DEPTH;

        SELECTORCOUNT = 1;
        INTATTRCOUNT  = 1;

        CHARATTRLENGTH = 0;
/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
*****/
        CALL MQINQ (HCONN,
                     HOBJ,
                     SELECTORCOUNT,
                     SELECTORS,
                     INTATTRCOUNT,
                     INTATTRS,
                     CHARATTRLENGTH,
                     CHARATTRS,
                     COMPCODE,
                     REASON);

/*****
/* TEST THE COMPLETION CODE OF THE INQUIRE CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING
/* THE COMPLETION CODE AND THE REASON CODE.
*****/
        IF COMPCODE = MQCC_OK
            THEN DO;
        :

```

```

        CALL ERROR_ROUTINE;
    END;

```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue.

This extract is not taken from the sample applications supplied with IBM MQ.

```

        %INCLUDE SYSLIB(CMQP);
        %INCLUDE SYSLIB(CMQEPP);
        :
/*****
/* WORKING STORAGE DECLARATIONS
*****/
        DCL COMPCODE          BINARY FIXED (31);
        DCL REASON            BINARY FIXED (31);
        DCL HCONN             BINARY FIXED (31);
        DCL HOBJ              BINARY FIXED (31);
        DCL OPTIONS           BINARY FIXED (31);
        DCL SELECTORCOUNT    BINARY FIXED (31);
        DCL INTATTRCOUNT    BINARY FIXED (31);
        DCL 1 SELECTOR_TABLE,
           3 SELECTORS(5)      BINARY FIXED (31);
        DCL 1 INTATTR_TABLE,
           3 INTATTRS(5)      BINARY FIXED (31);
        DCL CHARATTRLENGTH    BINARY FIXED (31);
        DCL CHARATTRS         CHAR(100);
        :

/*****
/* SET VARIABLES FOR SET CALL
/* SET GET AND PUT INHIBITED
*****/
        SELECTORS(01) = MQIA_INHIBIT_GET;
        SELECTORS(02) = MQIA_INHIBIT_PUT;

        INTATTRS(01) = MQQA_GET_INHIBITED;
        INTATTRS(02) = MQQA_PUT_INHIBITED;

        SELECTORCOUNT = 2;
        INTATTRCOUNT  = 2;

        CHARATTRLENGTH = 0;

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
/*
*****/
        CALL MQSET (HCONN,
                     HOBJ,
                     SELECTORCOUNT,
                     SELECTORS,
                     INTATTRCOUNT,
                     INTATTRS,
                     CHARATTRLENGTH,
                     CHARATTRS,
                     COMPCODE,
                     REASON);

/*****
/* TEST THE COMPLETION CODE OF THE SET CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING
/* THE COMPLETION CODE AND THE REASON CODE.
*****/
        IF COMPCODE = MQCC_OK
        THEN DO;
            :
            CALL ERROR_ROUTINE;
        END;

```

Constants

Use the reference information in this section to accomplish the tasks that address your business needs.

IBM MQ COPY, header, include, and module files

This information is general-use programming interface information.

This section contains information to help you use the MQI for various programming languages, as follows.

C header files

Header files are provided to help you write C application programs that use the MQI. These header files are summarized in the table:

Table 1. C header files - call prototypes, data types, return codes, constants, and structures					
File name	Description	IBM i	UNIX and Linux® systems	Windows	z/OS
Call prototypes, data types, return codes, constants, and structures					
CMQC	MQI definitions	C	C	C	C
CMQBC	MQAI definitions	C	C	C	
CMQEC	Interface Entry Points definition (includes CMQC, CMQXC and CMQZC)		C	C	
CMQCFC	PCF definitions	C	C	C	C
CMQPSC	Publish/Subscribe definitions	C	C	C	C
CMQXC	Channel and exit definitions	C	C	C	C
CMQZC	Installable services definitions	C	C	C	
Key: C= Files provided					

COBOL COPY files

Various COPY files are provided to help you write COBOL application programs that use the MQI. These files are summarized in the table:

Table 2. COBOL copy files - return codes, constants, and structures					
File name	Description	IBM i	UNIX systems	Windows	z/OS
Return codes and constants					
CMQx	MQI definitions	V	V	V	V
CMQCFx	PCF definitions	V	V	V	V
CMQPSx	Publish/Subscribe definitions	V	V	V	V
CMQXx	Channel and exit definitions	V	V	V	V
Structures					
CMQAIRx	MQAIR - Authentication information record		V L	V L	
CMQBOx	MQBO - Begin options	V L	V L	V L	
CMQCDx	MQCD - Channel definition	V L	V L	V L	V L

Table 2. COBOL copy files - return codes, constants, and structures (continued)

File name	Description	IBM i	UNIX systems	Windows	z/OS
CMQCFBFx	MQCFBF - PCF byte string filter parameter	V L	V L	V L	V L
CMQCFBSx	MQCFBS - PCF byte string parameter	V L	V L	V L	V L
CMQCFGRx	MQCFGR - PCF group parameter	V L	V L	V L	V L
CMQCFHx	MQCFH - PCF header	V L	V L	V L	V L
CMQCFIFx	MQCFIF - PCF integer filter parameter	V L	V L	V L	V L
CMQCFILx	MQCFIL - PCF integer list parameter	V L	V L	V L	V L
CMQCFINx	MQCFIN - PCF integer parameter	V L	V L	V L	V L
CMQCFSFx	MQCFSF - PCF string filter parameter	V L	V L	V L	V L
CMQCFSLx	MQCFSL - PCF string list parameter	V L	V L	V L	V L
CMQCFSTx	MQCFST - PCF string parameter	V L	V L	V L	V L
CMQCFXLx	MQCFIL64 - PCF 64-bit integer list parameter	V L	V L	V L	V L
CMQCFXNx	MQCFIN64 - PCF 64-bit integer parameter	V L	V L	V L	V L
CMQCHRVx	MQCHARV - Variable length string	V L	V L	V L	V L
CMQCIHx	MQCIH - CICS® bridge header	V L	V L	V L	V L
CMQCNOx	MQCNO - Connect options	V L	V L	V L	V L
CMQCSPx	MQCSP - Security parameters	V L	V L	V L	V L
CMQCXPx	MQCXP - Channel exit parameters	V L			V L
CMQDHx	MQDH - Distribution header	V L	V L	V L	V L
CMQDLHx	MQDLH - Dead-letter header	V L	V L	V L	V L
CMQDXPx	MQDXP - Data conversion exit parameters	V L		V L	
CMQEPHx	MQEPH - Embedded PCF header	V L	V L	V L	V L
CMQGMOx	MQGMO - Get message options	V L	V L	V L	V L
CMQIIHx	MQIIH - IMS information header	V L	V L	V L	V L
CMQMDx	MQMD - Message descriptor	V L	V L	V L	V L
CMQMD1x	MQMD1 - Message descriptor version 1	V L	V L	V L	V L
CMQMD2x	MQMD2 - Message descriptor version 2	V L	V L	V L	V L
CMQMDEx	MQMDE - Message descriptor extended	V L	V L	V L	V L

Table 2. COBOL copy files - return codes, constants, and structures (continued)

File name	Description	IBM i	UNIX systems	Windows	z/OS
CMQODx	MQOD - Object descriptor	V L	V L	V L	V L
CMQORx	MQOR - Object record	V L	V L	V L	V L
CMQPMOx	MQPMO - Put message options	V L	V L	V L	V L
CMQRFHx	MQRFH - Rules and formatting header	V L	V L	V L	V L
CMQRFH2x	MQRFH2 - Rules and formatting header 2	V L	V L	V L	V L
CMQRMHx	MQRMH - Reference message header	V L	V L	V L	V L
CMQRRx	MQRR - Response record	V L	V L	V L	
CMQSCOx	MQSCO - SSL configuraton options		V L	V L	
CMQTMx	MQTM - Trigger message	V L		V L	V L
CMQTCx	MQTMC - Trigger message character	V L	V L		
CMQTC2x	MQTMC2 - Trigger message 2 character	V L	V L	V L	V L
CMQWIHx	MQWIH - Work information header	V L	V L	V L	V L
CMQXQHx	MQXQH - Transmission queue header	V L	V L	V L	V L
Key: <ul style="list-style-type: none"> Files with initial values provided, x=V Files without initial values provided, x=L 					

PL/I include files

The following INCLUDE files are provided for the PL/I programming language. These files are available on z/OS only.

Table 3. PL/I include files - data types, return codes, constants, and structures

File name	Description	IBM i	UNIX systems	Windows	z/OS
Data types, return codes, constants, and structures					
CMQP	MQI definitions				P
CMQCFP	PCF definitions				P
CMQEPP	Entry point definitions				P
CMQPSP	Publish/Subscribe definitions				P
CMQXP	Channel and exit definitions				P
Key: P= File provided					

RPG copy files

The following COPY files are provided for the RPG programming language. These files are available only on IBM i.

Table 4. RPG copy files - return codes, constants, and structures					
File name	Description	IBM i	UNIX systems	Windows	z/OS
Return codes and constants					
CMQx	MQI definitions	G R			
CMQCFx	PCF definitions	G			
CMQPSx	Publish/Subscribe definitions	G			
CMQXx	Channel and exit definitions	G R			
Structures					
CMQBOx	MQBO - Begin options	G H			
CMQCDx	MQCD - Channel definition	G H R			
CMQCFBFx	MQCFBF - PCF byte string filter parameter	G H			
CMQCFBSx	MQCFBS - PCF byte string parameter	G H			
CMQCFGRx	MQCFGR - PCF group parameter	G H			
CMQCFHx	MQCFH - PCF header	G H			
CMQCFIFx	MQCFIF - PCF integer filter parameter	G H			
CMQCFILx	MQCFIL - PCF integer list parameter	G H			
CMQCFINx	MQCFIN - PCF integer parameter	G H			
CMQCFSFx	MQCFSF - PCF string filter parameter	G H			
CMQCFSLx	MQCFSL - PCF string list parameter	G H			
CMQCFSTx	MQCFST - PCF string parameter	G H			
CMQCFXLx	MQCFIL64 - PCF 64-bit integer list parameter	G H			
CMQCFXNx	MQCFIN64 - PCF 64-bit integer parameter	G H			
CMQCHARVx	MQCHARV - Variable length string	G H			
CMQCIHx	MQCIH - CICS bridge header	G H			
CMQCNOx	MQCNO - Connect options	G H			
CMQCSPx	MQCSP - Security parameters	G H			
CMQCXPx	MQCXP - Channel exit parameters	G H R			
CMQDHx	MQDH - Distribution header	G H R			
CMQDLHx	MQDLH - Dead-letter header	G H R			

Table 4. RPG copy files - return codes, constants, and structures (continued)

File name	Description	IBM i	UNIX systems	Windows	z/OS
CMQDXPx	MQDXP - Data conversion exit parameters	G H R			
CMQEPHx	MQEPH - Embedded PCF header	G H			
CMQGMox	MQGMO - Get message options	G H R			
CMQIIHx	MQIIH - IMS information header	G H R			
CMQMdx	MQMD - Message descriptor	G H R			
CMQMdx1x	MQMD1 - Message descriptor version 1	G H R			
CMQMdx2x	MQMD2 - Message descriptor version 2	G H			
CMQMDEx	MQMDE - Message descriptor extended	G H R			
CMQODx	MQOD - Object descriptor	G H R			
CMQORx	MQOR - Object record	G H R			
CMQPMox	MQPMO - Put message options	G H R			
CMQPPPx	MQPPX - Publish/Subscribe routing exit parameters	G H			
CMQRFHx	MQRFH - Rules and formatting header	G H			
CMQRFH2x	MQRFH2 - Rules and formatting header 2	G H			
CMQRMHx	MQRMH - Reference message header	G H R			
CMQRRx	MQRR - Response record	G H R			
CMQTMx	MQTM - Trigger message	G H R			
CMQTMcx	MQTMC - Trigger message character	G H R			
CMQTMcx2x	MQTMC2 - Trigger message 2 character	G H R			
CMQWIHx	MQWIH - Work information header	G H			
CMQXQHx	MQXQH - Transmission queue header	G H R			

Key:

- File for static linkage, initialized, provided x=G
- File for static linkage, not initialized, provided x=H
- File for dynamic linkage, initialized, provided, x=R

Visual Basic module files

Header (or form) files are provided to help you write Visual Basic application programs that use the MQI. These header files are supplied in 32-bit versions only and are summarized in the table:

Table 5. Visual Basic module files - call declarations, data types, return codes, constants, and structures

File name	Description	IBM i	UNIX and Linux systems	Windows	z/OS
Call declarations, data types, return codes, constants, and structures					
CMQB	MQI definitions			B	
CMQBB	MQAI definitions			B	
CMQCFB	PCF definitions			B	
CMQXB	Channel and exit definitions			B	
Key: B= File provided					

z/OS Assembler COPY files

Various COPY files are provided to help you write z/OS Assembler application programs that use the MQI. These files are summarized in the table:

Table 6. z/OS Assembler copy files - data types, return codes, constants, and structures

File name	Description	IBM i	UNIX systems	Windows	z/OS
Data types, return codes, and constants					
CMQA	MQI definitions				A
CMQCFA	PCF definitions				A
CMQPSA	Publish/Subscribe definitions				A
CMQVERA	Structure version control				A
CMQXA	Channel and exit definitions				A
Structures					
CMQCDA	MQCD - Channel definition				
CMQCFBFA	MQCFBF - PCF byte string filter parameter				
CMQCFBSA	MQCFBS - PCF byte string parameter				A
CMQCFGRA	MQCFGR - PCF group parameter				A
CMQCFHA	MQCFH - PCF header				A
CMQCFIFA	MQCFIF - PCF integer filter parameter				A
CMQCFILA	MQCFIL - PCF integer list parameter				A
CMQCFINA	MQCFIN - PCF integer parameter				A
CMQCFSTA	MQCFST - PCF string parameter				A
CMQCFSLA	MQCFSL - PCF string list parameter				A
CMQCFSTA	MQCFST - PCF string parameter				A

Table 6. z/OS Assembler copy files - data types, return codes, constants, and structures (continued)

File name	Description	IBM i	UNIX systems	Windows	z/OS
CMQCFXLA	MQCFIL64 - PCF 64-bit integer list parameter				A
CMQCFXNA	MQCFIN64 - PCF 64-bit integer parameter				A
CMQCHARVA	MQCHARV - Variable length string				A
CMQCIHA	MQCIH - CICS bridge header				A
CMQCNOA	MQCNO - Connect options				A
CMQCSPA	MQCSP - Security parameters				A
CMQCXPA	MQCXP - Channel exit parameters				A
CMQDHA	MQDH - Distribution header				A
CMQDLHA	MQDLH - Dead-letter header				A
CMQDXPA	MQDXP - Data conversion exit parameters				A
CMQEPHA	MQEPH - Embedded PCF header				A
CMQGMOA	MQGMO - Get message options				A
CMQIIHA	MQIIH - IMS information header				A
CMQMDA	MQMD - Message descriptor				A
CMQMD1A	MQMD1 - Message descriptor version 1				A
CMQMD2A	MQMD2 - Message descriptor version 2				A
CMQMDEA	MQMDE - Message descriptor extended				A
CMQODA	MQOD - Object descriptor				A
CMQORA	MQOR - Object record				A
CMQPMOA	MQPMO - Put message options				A
CMQRFHA	MQRFH - Rules and formatting header				A
CMQRFH2A	MQRFH2 - Rules and formatting header 2				A
CMQRMHA	MQRMH - Reference message header				A
CMQTMA	MQTM - Trigger message				A
CMQTMC2A	MQTMC2 - Trigger message 2 character				A
CMQWCRA	MQWCR - Cluster workload cluster record				A

Table 6. z/OS Assembler copy files - data types, return codes, constants, and structures (continued)

File name	Description	IBM i	UNIX systems	Windows	z/OS
CMQWDRA	MQWDR - Cluster workload destination record				A
CMQWDR1A	MQWDR1 - Cluster workload destination record version 1				A
CMQWDR2A	MQWDR2 - Cluster workload destination record version 2				A
CMQWIHA	MQWIH - Work information header				A
CMQWQRA	MQWQR - Cluster workload queue record				A
CMQWQR1A	MQWQR1 - Cluster workload queue record version 1				A
CMQWQR2A	MQWQR2 - Cluster workload queue record version 2				A
CMQWXP	MQWXP - Cluster workload exit parameters				A
CMQWXP1A	MQWXP1 - Cluster workload exit parameters version 1				A
CMQWXP2A	MQWXP2 - Cluster workload exit parameters version 2				A
CMQWXP3A	MQWXP3 - Cluster workload exit parameters version 3				A
CMQXPA	MQXP - CICS API-crossing exit parameters				A
CMQXQHA	MQXQH - Transmission queue header				A
CMQXWDA	MQXWD - Exit wait descriptor				A
Key: A= File provided					

MQ_* (String Lengths)

Table 7. Values of constants

Name	Decimal value	Hexadecimal value
MQ_ABEND_CODE_LENGTH	4	X'00000004'
MQ_ACCOUNTING_TOKEN_LENGTH	32	X'00000020'
MQ_APPL_IDENTITY_DATA_LENGTH	32	X'00000020'
MQ_APPL_NAME_LENGTH	28	X'0000001C'
MQ_APPL_ORIGIN_DATA_LENGTH	4	X'00000004'
MQ_APPL_TAG_LENGTH	28	X'0000001C'
MQ_ARM_SUFFIX_LENGTH	2	X'00000002'
MQ_ATTENTION_ID_LENGTH	4	X'00000004'

Table 7. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQ_AUTH_INFO_CONN_NAME_LENGTH	264	X'00000108'
MQ_AUTH_INFO_DESC_LENGTH	64	X'00000040'
MQ_AUTH_INFO_NAME_LENGTH	48	X'00000030'
MQ_AUTH_INFO_OCSP_URL_LENGTH	256	X'00000100'
MQ_AUTHENTICATOR_LENGTH	8	X'00000008'
MQ_AUTO_REORG_CATALOG_LENGTH	44	X'0000002C'
MQ_AUTO_REORG_TIME_LENGTH	4	X'00000004'
MQ_BATCH_INTERFACE_ID_LENGTH	8	X'00000008'
MQ_BRIDGE_NAME_LENGTH	24	X'00000018'
MQ_CANCEL_CODE_LENGTH	4	X'00000004'
MQ_CF_STRUC_DESC_LENGTH	64	X'00000040'
MQ_CF_STRUC_NAME_LENGTH	12	X'0000000C'
MQ_CHANNEL_DATE_LENGTH	12	X'0000000C'
MQ_CHANNEL_DESC_LENGTH	64	X'00000040'
MQ_CHANNEL_NAME_LENGTH	20	X'00000014'
MQ_CHANNEL_TIME_LENGTH	8	X'00000008'
MQ_CHINIT_SERVICE_PARM_LENGTH	32	X'00000020'
MQ_CICS_FILE_NAME_LENGTH	8	X'00000008'
MQ_CLIENT_ID_LENGTH	23	X'00000017'
MQ_CLUSTER_NAME_LENGTH	48	X'00000030'
MQ_CONN_NAME_LENGTH	264	X'00000108'
MQ_CONN_TAG_LENGTH	128	X'00000080'
MQ_CONNECTION_ID_LENGTH	24	X'00000018'
MQ_CORREL_ID_LENGTH	24	X'00000018'
MQ_CREATION_DATE_LENGTH	12	X'0000000C'
MQ_CREATION_TIME_LENGTH	8	X'00000008'
MQ_DATE_LENGTH	12	X'0000000C'
MQ_DISTINGUISHED_NAME_LENGTH	1024	X'00000400'
MQ_DNS_GROUP_NAME_LENGTH	18	X'00000012'
MQ_EXIT_DATA_LENGTH	32	X'00000020'
MQ_EXIT_INFO_NAME_LENGTH	48	X'00000030'
MQ_EXIT_NAME_LENGTH	(value differs by platform or version)	
MQ_EXIT_PD_AREA_LENGTH	48	X'00000030'
MQ_EXIT_USER_AREA_LENGTH	16	X'00000010'
MQ_FACILITY_LENGTH	8	X'00000008'
MQ_FACILITY_LIKE_LENGTH	4	X'00000004'
MQ_FORMAT_LENGTH	8	X'00000008'
MQ_FUNCTION_LENGTH	4	X'00000004'
MQ_GROUP_ID_LENGTH	24	X'00000018'

Table 7. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQ_LDAP_PASSWORD_LENGTH	32	X'00000020'
MQ_LISTENER_NAME_LENGTH	48	X'00000030'
MQ_LISTENER_DESC_LENGTH	64	X'00000040'
MQ_LOCAL_ADDRESS_LENGTH	48	X'00000030'
MQ_LTERM_OVERRIDE_LENGTH	8	X'00000008'
MQ_LU_NAME_LENGTH	8	X'00000008'
MQ_LUWID_LENGTH	16	X'00000010'
MQ_MAX_EXIT_NAME_LENGTH	128	X'00000080'
MQ_MAX_MCA_USER_ID_LENGTH	64	X'00000040'
MQ_MAX_PROPERTY_NAME_LENGTH	4095	X'0000FFFF'
MQ_MAX_USER_ID_LENGTH	64	X'00000040'
MQ_MCA_JOB_NAME_LENGTH	28	X'0000001C'
MQ_MCA_NAME_LENGTH	20	X'00000014'
MQ_MCA_USER_DATA_LENGTH	32	X'00000020'
MQ_MCA_USER_ID_LENGTH	(value differs by platform or version)	
MQ_MFS_MAP_NAME_LENGTH	8	X'00000008'
MQ_MODE_NAME_LENGTH	8	X'00000008'
MQ_MSG_HEADER_LENGTH	4000	X'00000FA0'
MQ_MSG_ID_LENGTH	24	X'00000018'
MQ_MSG_TOKEN_LENGTH	16	X'00000010'
MQ_NAMELIST_DESC_LENGTH	64	X'00000040'
MQ_NAMELIST_NAME_LENGTH	48	X'00000030'
MQ_OBJECT_INSTANCE_ID_LENGTH	24	X'00000018'
MQ_OBJECT_NAME_LENGTH	48	X'00000030'
MQ_PASS_TICKET_APPL_LENGTH	8	X'00000008'
MQ_PASSWORD_LENGTH	12	X'0000000C'
MQ_PROCESS_APPL_ID_LENGTH	256	X'00000100'
MQ_PROCESS_DESC_LENGTH	64	X'00000040'
MQ_PROCESS_ENV_DATA_LENGTH	128	X'00000080'
MQ_PROCESS_NAME_LENGTH	48	X'00000030'
MQ_PROCESS_USER_DATA_LENGTH	128	X'00000080'
MQ_PROGRAM_NAME_LENGTH	20	X'00000014'
MQ_PUT_APPL_NAME_LENGTH	28	X'0000001C'
MQ_PUT_DATE_LENGTH	8	X'00000008'
MQ_PUT_TIME_LENGTH	8	X'00000008'
MQ_Q_DESC_LENGTH	64	X'00000040'
MQ_Q_MGR_DESC_LENGTH	64	X'00000040'
MQ_Q_MGR_IDENTIFIER_LENGTH	48	X'00000030'
MQ_Q_MGR_NAME_LENGTH	48	X'00000030'

Table 7. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQ_Q_NAME_LENGTH	48	X'00000030'
MQ_QSG_NAME_LENGTH	4	X'00000004'
MQ_REMOTE_SYS_ID_LENGTH	4	X'00000004'
MQ_SECURITY_ID_LENGTH	40	X'00000028'
MQ_SELECTOR_LENGTH	10240	X'00002800'
MQ_SERVICE_ARGS_LENGTH	255	X'000000FF'
MQ_SERVICE_COMMAND_LENGTH	255	X'000000FF'
MQ_SERVICE_DESC_LENGTH	64	X'00000040'
MQ_SERVICE_NAME_LENGTH	32	X'00000020'
MQ_SERVICE_PATH_LENGTH	255	X'000000FF'
MQ_SERVICE_STEP_LENGTH	8	X'00000008'
MQ_SHORT_CONN_NAME_LENGTH	20	X'00000014'
MQ_SHORT_DNAME_LENGTH	256	X'00000100'
MQ_SSL_CIPHER_SPEC_LENGTH	32	X'00000020'
MQ_SSL_CRYPTOHARDWARE_LENGTH	256	X'00000100'
MQ_SSL_HANDSHAKE_STAGE_LENGTH	32	X'00000020'
MQ_SSL_KEY_LIBRARY_LENGTH	44	X'0000002C'
MQ_SSL_KEY_MEMBER_LENGTH	8	X'00000008'
MQ_SSL_KEY_REPOSITORY_LENGTH	256	X'00000100'
MQ_SSL_PEER_NAME_LENGTH	1024	X'00000400'
MQ_SSL_SHORT_PEER_NAME_LENGTH	256	X'00000100'
MQ_START_CODE_LENGTH	4	X'00000004'
MQ_STORAGE_CLASS_DESC_LENGTH	64	X'00000040'
MQ_STORAGE_CLASS_LENGTH	8	X'00000008'
MQ_SUB_IDENTITY_LENGTH	128	X'00000080'
MQ_SUB_POINT_LENGTH	128	X'00000080'
MQ_SUITE_B_128_BIT	2	X'00000002'
MQ_SUITE_B_192_BIT	4	X'00000004'
MQ_SUITE_B_NONE	1	X'00000001'
MQ_SUITE_B_NOT_AVAILABLE	0	X'00000000'
MQ_TCP_NAME_LENGTH	8	X'00000008'
MQ_TIME_LENGTH	8	X'00000008'
MQ_TOPIC_DESC_LENGTH	64	X'00000040'
MQ_TOPIC_NAME_LENGTH	48	X'00000030'
MQ_TOPIC_STR_LENGTH	10240	X'00002800'
MQ_TOTAL_EXIT_DATA_LENGTH	999	X'000003E7'
MQ_TOTAL_EXIT_NAME_LENGTH	999	X'000003E7'
MQ_TP_NAME_LENGTH	64	X'00000040'
MQ_TPIPE_NAME_LENGTH	8	X'00000008'

Table 7. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQ_TRAN_INSTANCE_ID_LENGTH	16	X'00000010'
MQ_TRANSACTION_ID_LENGTH	4	X'00000004'
MQ_TRIGGER_DATA_LENGTH	64	X'00000040'
MQ_TRIGGER_PROGRAM_NAME_LENGTH	8	X'00000008'
MQ_TRIGGER_TERM_ID_LENGTH	4	X'00000004'
MQ_TRIGGER_TRANS_ID_LENGTH	4	X'00000004'
MQ_USER_ID_LENGTH	12	X'0000000C'
MQ_VERSION_LENGTH	8	X'00000008'
MQ_XCF_GROUP_NAME_LENGTH	8	X'00000008'
MQ_XCF_MEMBER_NAME_LENGTH	16	X'00000010'

MQ_* (Command format String Lengths)


Table 8. Values of constants		
Name	Decimal value	Hexadecimal value
MQ_ARCHIVE_PFX_LENGTH	36	X'00000024'
MQ_ARCHIVE_UNIT_LENGTH	8	X'00000008'
MQ_ASID_LENGTH	4	X'00000004'
MQ_AUTH_PROFILE_NAME_LENGTH	48	X'00000030'
MQ_CF_LEID_LENGTH	12	X'0000000C'
MQ_COMMAND_MQSC_LENGTH	32768	X'00008000'
MQ_DATA_SET_NAME_LENGTH	44	X'0000002C'
MQ_DB2_NAME_LENGTH	4	X'00000004'
MQ_DSG_NAME_LENGTH	8	X'00000008'
 MQ_ENTITY_NAME_LENGTH	1024	X'00000400'
MQ_ENV_INFO_LENGTH	96	X'00000060'
MQ_IP_ADDRESS_LENGTH	48	X'00000030'
MQ_LOG_CORREL_ID_LENGTH	8	X'00000008'
MQ_LOG_EXTENT_NAME_LENGTH	24	X'00000018'
MQ_LOG_PATH_LENGTH	1024	X'00000400'
MQ_LRSN_LENGTH	12	X'0000000C'
MQ_ORIGIN_NAME_LENGTH	8	X'00000008'
MQ_PSB_NAME_LENGTH	8	X'00000008'
MQ_PST_ID_LENGTH	8	X'00000008'
MQ_Q_MGR_CPF_LENGTH	4	X'00000004'
MQ_RESPONSE_ID_LENGTH	24	X'00000018'
MQ_RBA_LENGTH	16	X'00000010'
MQ_SECURITY_PROFILE_LENGTH	40	X'00000028'
MQ_SERVICE_COMPONENT_LENGTH	48	X'00000030'
MQ_SUB_NAME_LENGTH	10240	X'00002800'

Table 8. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQ_SYSP_SERVICE_LENGTH	32	X'00000020'
MQ_SYSTEM_NAME_LENGTH	8	X'00000008'
MQ_TASK_NUMBER_LENGTH	8	X'00000008'
MQ_TPIPE_PFX_LENGTH	4	X'00000004'
MQ_UOW_ID_LENGTH	256	X'00000100'
MQ_USER_DATA_LENGTH	10240	X'00002800'
MQ_VOLSER_LENGTH	6	X'00000006'

MQACH_* (API exit chain area header structure)

Table 9. Structures of constants	
Name	Structure
MQACH_STRUC_ID	"ACH¬"
MQACH_STRUC_ID_ARRAY	'A','C','H','¬'

Note: The symbol ¬ represents a single blank character.

Table 10. Values of constants		
Name	Decimal value	Hexadecimal value
MQACH_VERSION_1	1	X'00000001'
MQACH_CURRENT_VERSION	1	X'00000001'
MQACH_LENGTH_1	(value differs by platform or version)	
MQACH_CURRENT_LENGTH	(value differs by platform or version)	

MQACT_* (Accounting Token)

Table 11. Constant names and values	
Name	Value
MQACT_NONE	X'00...00' (32 nulls)
MQACT_NONE_ARRAY	'\0','\0',... (32 nulls)

MQACT_* (Command format Action Options)

Table 12. Values of constants		
Name	Decimal value	Hexadecimal value
MQACT_FORCE_REMOVE	1	X'00000001'
MQACT_ADVANCE_LOG	2	X'00000002'
MQACT_COLLECT_STATISTICS	3	X'00000003'
MQACT_PUBSUB	4	X'00000004'

MQACTP_* (Action)

Table 13. Values of constants		
Name	Decimal value	Hexadecimal value
MQACTP_NEW	0	X'00000000'
MQACTP_FORWARD	1	X'00000001'
MQACTP_REPLY	2	X'00000002'
MQACTP_REPORT	3	X'00000003'

MQACTT_* (Accounting Token Types)

Table 14. Values of constants	
Name	Hexadecimal value
MQACTT_UNKNOWN	X'00'
MQACTT_CICS_LUOW_ID	X'01'
MQACTT_OS2_DEFAULT	X'04'
MQACTT_DOS_DEFAULT	X'05'
MQACTT_UNIX_NUMERIC_ID	X'06'
MQACTT_OS400_ACCOUNT_TOKEN	X'08'
MQACTT_WINDOWS_DEFAULT	X'09'
MQACTT_NT_SECURITY_ID	X'0B'
MQACTT_USER	X'19'

MQADOPT_* (Adopt New MCA Checks and Adopt New MCA Types)

Adopt New MCA Checks

Table 15. Values of constants		
Name	Decimal value	Hexadecimal value
MQADOPT_CHECK_NONE	0	X'00000000'
MQADOPT_CHECK_ALL	1	X'00000001'
MQADOPT_CHECK_Q_MGR_NAME	2	X'00000002'
MQADOPT_CHECK_NET_ADDR	4	X'00000004'

Adopt New MCA Types

Table 16. Values of constants		
Name	Decimal value	Hexadecimal value
MQADOPT_TYPE_NO	0	X'00000000'
MQADOPT_TYPE_ALL	1	X'00000001'
MQADOPT_TYPE_SVR	2	X'00000002'
MQADOPT_TYPE_SDR	4	X'00000004'
MQADOPT_TYPE_RCVR	8	X'00000008'
MQADOPT_TYPE_CLUSRCVR	16	X'00000010'

MQAIR_* (Authentication information record structure)

Table 17. Structures of constants	
Name	Structure
MQAIR_STRUC_ID	"AIR↵"
MQAIR_STRUC_ID_ARRAY	'A','I','R','↵'

Note: The symbol ↵ represents a single blank character.

Table 18. Values of constants		
Name	Decimal value	Hexadecimal value
MQAIR_VERSION_1	1	X'00000001'
MQAIR_VERSION_2	2	X'00000002'
MQAIR_CURRENT_VERSION	2	X'00000002'

MQAIT_* (Authentication Information Type)

Table 19. Values of constants		
Name	Decimal value	Hexadecimal value
MQAIT_ALL	0	X'00000000'
MQAIT_CRL_LDAP	1	X'00000001'
MQAIT_OCSP	2	X'00000002'
MQAIT_IDPW_OS	3	X'00000003'
MQAIT_IDPW_LDAP	4	X'00000004'

MQAS_* (Command format Asynchronous State Values)

Table 20. Values of constants		
Name	Decimal value	Hexadecimal value
MQAS_NONE	0	X'00000000'
MQAS_STARTED	1	X'00000001'
MQAS_START_WAIT	2	X'00000002'
MQAS_STOPPED	3	X'00000003'
MQAS_SUSPENDED	4	X'00000004'
MQAS_SUSPENDED_TEMPORARY	5	X'00000005'
MQAS_ACTIVE	6	X'00000006'
MQAS_INACTIVE	7	X'00000007'

MQAT_* (Put Application Types)

Table 21. Values of constants		
Name	Decimal value	Hexadecimal value
MQAT_UNKNOWN	-1	X'FFFFFFFF'
MQAT_NO_CONTEXT	0	X'00000000'
MQAT_CICS	1	X'00000001'
MQAT_MVS	2	X'00000002'
MQAT_OS390	2	X'00000002'

Table 21. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQAT_ZOS	2	X'00000002'
MQAT_IMS	3	X'00000003'
MQAT_OS2	4	X'00000004'
MQAT_DOS	5	X'00000005'
MQAT_AIX®	6	X'00000006'
MQAT_UNIX	6	X'00000006'
MQAT_QMGR	7	X'00000007'
MQAT_OS400	8	X'00000008'
MQAT_WINDOWS	9	X'00000009'
MQAT_CICS_VSE	10	X'0000000A'
MQAT_WINDOWS_NT	11	X'0000000B'
MQAT_VMS	12	X'0000000C'
MQAT_GUARDIAN	13	X'0000000D'
MQAT_NSK	13	X'0000000D'
MQAT_VOS	14	X'0000000E'
MQAT_OPEN_TP1	15	X'0000000F'
MQAT_VM	18	X'00000012'
MQAT_IMS_BRIDGE	19	X'00000013'
MQAT_XCF	20	X'00000014'
MQAT_CICS_BRIDGE	21	X'00000015'
MQAT_NOTES_AGENT	22	X'00000016'
MQAT_TPF	23	X'00000017'
MQAT_USER	25	X'00000019'
MQAT_BROKER	26	X'0000001A'
MQAT_QMGR_PUBLISH	26	X'0000001A'
MQAT_JAVA	28	X'0000001C'
MQAT_DQM	29	X'0000001D'
MQAT_CHANNEL_INITIATOR	30	X'0000001E'
MQAT_WLM	31	X'0000001F'
MQAT_BATCH	32	X'00000020'
MQAT_RRS_BATCH	33	X'00000021'
MQAT_SIB	34	X'00000022'
MQAT_DEFAULT	(value differs by platform or version)	
MQAT_USER_FIRST	65536	X'00010000'
MQAT_USER_LAST	999999999	X'3B9AC9FF'

MQAUTH_* (Command format Authority Values)

Table 22. Values of constants		
Name	Decimal value	Hexadecimal value
MQAUTH_NONE	0	X'00000000'
MQAUTH_ALT_USER_AUTHORITY	1	X'00000001'
MQAUTH_BROWSE	2	X'00000002'
MQAUTH_CHANGE	3	X'00000003'
MQAUTH_CLEAR	4	X'00000004'
MQAUTH_CONNECT	5	X'00000005'
MQAUTH_CREATE	6	X'00000006'
MQAUTH_DELETE	7	X'00000007'
MQAUTH_DISPLAY	8	X'00000008'
MQAUTH_INPUT	9	X'00000009'
MQAUTH_INQUIRE	10	X'0000000A'
MQAUTH_OUTPUT	11	X'0000000B'
MQAUTH_PASS_ALL_CONTEXT	12	X'0000000C'
MQAUTH_PASS_IDENTITY_CONTEXT	13	X'0000000D'
MQAUTH_SET	14	X'0000000E'
MQAUTH_SET_ALL_CONTEXT	15	X'0000000F'
MQAUTH_SET_IDENTITY_CONTEXT	16	X'00000010'
MQAUTH_CONTROL	17	X'00000011'
MQAUTH_CONTROL_EXTENDED	18	X'00000012'
MQAUTH_PUBLISH	19	X'00000013'
MQAUTH_SUBSCRIBE	20	X'00000014'
MQAUTH_RESUME	21	X'00000015'
MQAUTH_SYSTEM	22	X'00000016'

MQAUTHOPT_* (Command format Authority Options)

Table 23. Values of constants		
Name	Decimal value	Hexadecimal value
MQAUTHOPT_CUMULATIVE	256	X'00000100'
MQAUTHOPT_ENTITY_EXPLICIT	1	X'00000001'
MQAUTHOPT_ENTITY_SET	2	X'00000002'
MQAUTHOPT_NAME_ALL_MATCHING	32	X'00000020'
MQAUTHOPT_NAME_AS_WILDCARD	64	X'00000040'
MQAUTHOPT_NAME_EXPLICIT	16	X'00000010'

MQAXC_* (API exit context structure)

Table 24. Structures of constants	
Name	Structure
MQAXC_STRUC_ID	"AXC-"

Table 24. Structures of constants (continued)

Name	Structure
MQAXC_STRUC_ID_ARRAY	'A', 'X', 'C', '␣'

Note: The symbol \neg represents a single blank character.

Table 25. Values of constants

Name	Decimal value	Hexadecimal value
MQAXC_VERSION_1	1	X'00000001'
MQAXC_CURRENT_VERSION	1	X'00000001'

MQAXP_* (API exit parameter structure)

Table 26. Structures of constants

Name	Structure
MQAXP_STRUC_ID	"AXP \neg "
MQAXP_STRUC_ID_ARRAY	'A', 'X', 'P', ' \neg '

Note: The symbol \neg represents a single blank character.

Table 27. Values of constants

Name	Decimal value	Hexadecimal value
MQAXP_VERSION_1	1	X'00000001'
MQAXP_VERSION_2	2	X'00000002'
MQAXP_CURRENT_VERSION	2	X'00000002'

MQBA_* (Byte Attribute Selectors)

Table 28. Values of constants

Name	Decimal value	Hexadecimal value
MQBA_FIRST	6001	X'00001771'
MQBA_LAST	8000	X'00001F40'

MQBACF_* (Command format Byte Parameter Types)

Table 29. Values of constants

Name	Decimal value	Hexadecimal value
MQBACF_FIRST	7001	X'00001B59'
MQBACF_EVENT_ACCOUNTING_TOKEN	7001	X'00001B59'
MQBACF_EVENT_SECURITY_ID	7002	X'00001B5A'
MQBACF_RESPONSE_SET	7003	X'00001B5B'
MQBACF_RESPONSE_ID	7004	X'00001B5C'
MQBACF_EXTERNAL_UOW_ID	7005	X'00001B5D'
MQBACF_CONNECTION_ID	7006	X'00001B5E'
MQBACF_GENERIC_CONNECTION_ID	7007	X'00001B5F'
MQBACF_ORIGIN_UOW_ID	7008	X'00001B60'
MQBACF_Q_MGR_UOW_ID	7009	X'00001B61'

Table 29. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQBACF_ACCOUNTING_TOKEN	7010	X'00001B62'
MQBACF_CORREL_ID	7011	X'00001B63'
MQBACF_GROUP_ID	7012	X'00001B64'
MQBACF_MSG_ID	7013	X'00001B65'
MQBACF_CF_LEID	7014	X'00001B66'
MQBACF_DESTINATION_CORREL_ID	7015	X'00001B67'
MQBACF_SUB_ID	7016	X'00001B68'
MQBACF_LAST_USED	7016	X'00001B68'

MQBL_* (Buffer Length for mqAddString and mqSetString)

Table 30. Values of constants		
Name	Decimal value	Hexadecimal value
MQBL_NULL_TERMINATED	-1	X'FFFFFFFF'

MQBMHO_* (Buffer to message handle options and structure)

Buffer to message handle options structure

Table 31. Structures of constants	
Name	Structure
MQBMHO_STRUC_ID	"BMHO"
MQBMHO_STRUC_ID_ARRAY	'B', 'M', 'H', 'O'

Note: The symbol ~ represents a single blank character.

Table 32. Values of constants		
Name	Decimal value	Hexadecimal value
MQBMHO_VERSION_1	1	X'00000001'
MQBMHO_CURRENT_VERSION	1	X'00000001'

Buffer To Message Handle Options

Table 33. Values of constants		
Name	Decimal value	Hexadecimal value
MQBMHO_NONE	0	X'00000000'
MQBMHO_DELETE_PROPERTIES	1	X'00000001'

MQBND_* (Default Bindings)

Table 34. Values of constants		
Name	Decimal value	Hexadecimal value
MQBND_BIND_ON_OPEN	0	X'00000000'
MQBND_BIND_NOT_FIXED	1	X'00000001'
MQBND_BIND_ON_GROUP	2	X'00000002'

MQBO_* (Begin options and structure)

Begin options structure

Table 35. Structures of constants	
Name	Structure
MQBO_STRUC_ID	"B0¬¬"
MQBO_STRUC_ID_ARRAY	'B','0','¬','¬'

Note: The symbol ¬ represents a single blank character.

Table 36. Values of constants		
Name	Decimal value	Hexadecimal value
MQBO_VERSION_1	1	X'00000001'
MQBO_CURRENT_VERSION	1	X'00000001'

Begin Options

Table 37. Values of constants		
Name	Decimal value	Hexadecimal value
MQBO_NONE	0	X'00000000'

MQBT_* (Command format Bridge Types)

Table 38. Values of constants		
Name	Decimal value	Hexadecimal value
MQBT_OTMA	1	X'00000001'

MQCA_* (Character Attribute Selectors)

Table 39. Values of constants		
Name	Decimal value	Hexadecimal value
MQCA_ADMIN_TOPIC_NAME	2105	X'00000839'
MQCA_ALTERATION_DATE	2027	X'000007EB'
MQCA_ALTERATION_TIME	2028	X'000007EC'
MQCA_APPL_ID	2001	X'000007D1'
MQCA_AUTH_INFO_CONN_NAME	2053	X'00000805'
MQCA_AUTH_INFO_DESC	2046	X'000007FE'
MQCA_AUTH_INFO_NAME	2045	X'000007FD'
MQCA_AUTH_INFO_OCSP_URL	2109	X'0000083D'
MQCA_AUTO_REORG_CATALOG	2091	X'0000082B'
MQCA_AUTO_REORG_START_TIME	2090	X'0000082A'
MQCA_BACKOUT_REQ_Q_NAME	2019	X'000007E3'
MQCA_BASE_OBJECT_NAME	2002	X'000007D2'
MQCA_BASE_Q_NAME	2002	X'000007D2'
MQCA_BATCH_INTERFACE_ID	2068	X'00000814'
MQCA_CF_STRUC_DESC	2052	X'00000804'

Table 39. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCA_CF_STRUC_NAME	2039	X'000007F7'
MQCA_CHANNEL_AUTO_DEF_EXIT	2026	X'000007EA'
MQCA_CHILD	2101	X'00000835'
MQCA_CHINIT_SERVICE_PARM	2076	X'0000081C'
MQCA_CICS_FILE_NAME	2060	X'0000080C'
MQCA_CLUS_CHL_NAME	2124	X'0000084C'
MQCA_CLUSTER_DATE	2037	X'000007F5'
MQCA_CLUSTER_NAME	2029	X'000007ED'
MQCA_CLUSTER_NAMELIST	2030	X'000007EE'
MQCA_CLUSTER_Q_MGR_NAME	2031	X'000007EF'
MQCA_CLUSTER_TIME	2038	X'000007F6'
MQCA_CLUSTER_WORKLOAD_DATA	2034	X'000007F2'
MQCA_CLUSTER_WORKLOAD_EXIT	2033	X'000007F1'
MQCA_COMMAND_INPUT_Q_NAME	2003	X'000007D3'
MQCA_COMMAND_REPLY_Q_NAME	2067	X'00000813'
MQCA_CREATION_DATE	2004	X'000007D4'
MQCA_CREATION_TIME	2005	X'000007D5'
MQCA_DEAD_LETTER_Q_NAME	2006	X'000007D6'
MQCA_DEF_XMIT_Q_NAME	2025	X'000007E9'
MQCA_DNS_GROUP	2071	X'00000817'
MQCA_ENV_DATA	2007	X'000007D7'
MQCA_FIRST	2001	X'000007D1'
MQCA_IGQ_USER_ID	2041	X'000007F9'
MQCA_INITIATION_Q_NAME	2008	X'000007D8'
MQCA_LAST	4000	X'00000FA0'
MQCA_LAST_USED	2109	X'0000083D'
MQCA_LDAP_PASSWORD	2048	X'00000800'
MQCA_LDAP_USER_NAME	2047	X'000007FF'
MQCA_LU_GROUP_NAME	2072	X'00000818'
MQCA_LU_NAME	2073	X'00000819'
MQCA_LU62_ARM_SUFFIX	2074	X'0000081A'
MQCA_MODEL_DURABLE_Q	2096	X'00000830'
MQCA_MODEL_NON_DURABLE_Q	2097	X'00000831'
MQCA_MONITOR_Q_NAME	2066	X'00000812'
MQCA_NAMELIST_DESC	2009	X'000007D9'
MQCA_NAMELIST_NAME	2010	X'000007DA'
MQCA_NAMES	2020	X'000007E4'
MQCA_PARENT	2102	X'00000836'
MQCA_PASS_TICKET_APPL	2086	X'00000826'

Table 39. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCA_PROCESS_DESC	2011	X'000007DB'
MQCA_PROCESS_NAME	2012	X'000007DC'
MQCA_Q_DESC	2013	X'000007DD'
MQCA_Q_MGR_DESC	2014	X'000007DE'
MQCA_Q_MGR_IDENTIFIER	2032	X'000007F0'
MQCA_Q_MGR_NAME	2015	X'000007DF'
MQCA_Q_NAME	2016	X'000007E0'
MQCA_QSG_NAME	2040	X'000007F8'
MQCA_REMOTE_Q_MGR_NAME	2017	X'000007E1'
MQCA_REMOTE_Q_NAME	2018	X'000007E2'
MQCA_REPOSITORY_NAME	2035	X'000007F3'
MQCA_REPOSITORY_NAMELIST	2036	X'000007F4'
MQCA_RESUME_DATE	2098	X'00000832'
MQCA_RESUME_TIME	2099	X'00000833'
MQCA_SERVICE_DESC	2078	X'0000081E'
MQCA_SERVICE_NAME	2077	X'0000081D'
MQCA_SERVICE_START_ARGS	2080	X'00000820'
MQCA_SERVICE_START_COMMAND	2079	X'0000081F'
MQCA_SERVICE_STOP_ARGS	2082	X'00000822'
MQCA_SERVICE_STOP_COMMAND	2081	X'00000821'
MQCA_STDERR_DESTINATION	2084	X'00000824'
MQCA_STDOUT_DESTINATION	2083	X'00000823'
MQCA_SSL_CRL_NAMELIST	2050	X'00000802'
MQCA_SSL_CRYPTO_HARDWARE	2051	X'00000803'
MQCA_SSL_KEY_LIBRARY	2069	X'00000815'
MQCA_SSL_KEY_MEMBER	2070	X'00000816'
MQCA_SSL_KEY_REPOSITORY	2049	X'00000801'
MQCA_STORAGE_CLASS	2022	X'000007E6'
MQCA_STORAGE_CLASS_DESC	2042	X'000007FA'
MQCA_SYSTEM_LOG_Q_NAME	2065	X'00000811'
MQCA_TCP_NAME	2075	X'0000081B'
MQCA_TOPIC_DESC	2093	X'0000082D'
MQCA_TOPIC_NAME	2092	X'0000082C'
MQCA_TOPIC_STRING_FILTER	2108	X'0000083C'
MQCA_TOPIC_STRING	2094	X'0000082E'
MQCA_TPIPE_NAME	2085	X'00000825'
MQCA_TRIGGER_CHANNEL_NAME	2064	X'00000810'
MQCA_TRIGGER_DATA	2023	X'000007E7'
MQCA_TRIGGER_PROGRAM_NAME	2062	X'0000080E'

Table 39. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCA_TRIGGER_TERM_ID	2063	X'0000080F'
MQCA_TRIGGER_TRANS_ID	2061	X'0000080D'
MQCA_USER_DATA	2021	X'000007E5'
MQCA_USER_LIST	4000	X'00000FA0'
MQCA_VERSION	2120	X'00000848'
MQCA_XCF_GROUP_NAME	2043	X'000007FB'
MQCA_XCF_MEMBER_NAME	2044	X'000007FC'
MQCA_XMIT_Q_NAME	2024	X'000007E8'

MQCACF_* (Command format Character Parameter Types)

Table 40. Values of constants		
Name	Decimal value	Hexadecimal value
MQCACF_FIRST	3001	X'00000BB9'
MQCACF_FROM_Q_NAME	3001	X'00000BB9'
MQCACF_TO_Q_NAME	3002	X'00000BBA'
MQCACF_FROM_PROCESS_NAME	3003	X'00000BBB'
MQCACF_TO_PROCESS_NAME	3004	X'00000BBC'
MQCACF_FROM_NAMELIST_NAME	3005	X'00000BBD'
MQCACF_TO_NAMELIST_NAME	3006	X'00000BBE'
MQCACF_FROM_CHANNEL_NAME	3007	X'00000BBF'
MQCACF_TO_CHANNEL_NAME	3008	X'00000BC0'
MQCACF_FROM_AUTH_INFO_NAME	3009	X'00000BC1'
MQCACF_TO_AUTH_INFO_NAME	3010	X'00000BC2'
MQCACF_Q_NAMES	3011	X'00000BC3'
MQCACF_PROCESS_NAMES	3012	X'00000BC4'
MQCACF_NAMELIST_NAMES	3013	X'00000BC5'
MQCACF_ESCAPE_TEXT	3014	X'00000BC6'
MQCACF_LOCAL_Q_NAMES	3015	X'00000BC7'
MQCACF_MODEL_Q_NAMES	3016	X'00000BC8'
MQCACF_ALIAS_Q_NAMES	3017	X'00000BC9'
MQCACF_REMOTE_Q_NAMES	3018	X'00000BCA'
MQCACF_SENDER_CHANNEL_NAMES	3019	X'00000BCB'
MQCACF_SERVER_CHANNEL_NAMES	3020	X'00000BCC'
MQCACF_REQUESTER_CHANNEL_NAMES	3021	X'00000BCD'
MQCACF_RECEIVER_CHANNEL_NAMES	3022	X'00000BCE'
MQCACF_OBJECT_Q_MGR_NAME	3023	X'00000BCF'
MQCACF_APPL_NAME	3024	X'00000BD0'
MQCACF_USER_IDENTIFIER	3025	X'00000BD1'
MQCACF_AUX_ERROR_DATA_STR_1	3026	X'00000BD2'

Table 40. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCACF_AUX_ERROR_DATA_STR_2	3027	X'00000BD3'
MQCACF_AUX_ERROR_DATA_STR_3	3028	X'00000BD4'
MQCACF_BRIDGE_NAME	3029	X'00000BD5'
MQCACF_STREAM_NAME	3030	X'00000BD6'
MQCACF_TOPIC	3031	X'00000BD7'
MQCACF_PARENT_Q_MGR_NAME	3032	X'00000BD8'
MQCACF_CORREL_ID	3033	X'00000BD9'
MQCACF_PUBLISH_TIMESTAMP	3034	X'00000BDA'
MQCACF_STRING_DATA	3035	X'00000BDB'
MQCACF_SUPPORTED_STREAM_NAME	3036	X'00000BDC'
MQCACF_REG_TOPIC	3037	X'00000BDD'
MQCACF_REG_TIME	3038	X'00000BDE'
MQCACF_REG_USER_ID	3039	X'00000BDF'
MQCACF_CHILD_Q_MGR_NAME	3040	X'00000BE0'
MQCACF_REG_STREAM_NAME	3041	X'00000BE1'
MQCACF_REG_Q_MGR_NAME	3042	X'00000BE2'
MQCACF_REG_Q_NAME	3043	X'00000BE3'
MQCACF_REG_CORREL_ID	3044	X'00000BE4'
MQCACF_EVENT_USER_ID	3045	X'00000BE5'
MQCACF_OBJECT_NAME	3046	X'00000BE6'
MQCACF_EVENT_Q_MGR	3047	X'00000BE7'
MQCACF_AUTH_INFO_NAMES	3048	X'00000BE8'
MQCACF_EVENT_APPL_IDENTITY	3049	X'00000BE9'
MQCACF_EVENT_APPL_NAME	3050	X'00000BEA'
MQCACF_EVENT_APPL_ORIGIN	3051	X'00000BEB'
MQCACF_SUBSCRIPTION_NAME	3052	X'00000BEC'
MQCACF_REG_SUB_NAME	3053	X'00000BED'
MQCACF_SUBSCRIPTION_IDENTITY	3054	X'00000BEE'
MQCACF_REG_SUB_IDENTITY	3055	X'00000BEF'
MQCACF_SUBSCRIPTION_USER_DATA	3056	X'00000BF0'
MQCACF_REG_SUB_USER_DATA	3057	X'00000BF1'
MQCACF_APPL_TAG	3058	X'00000BF2'
MQCACF_DATA_SET_NAME	3059	X'00000BF3'
MQCACF_UOW_START_DATE	3060	X'00000BF4'
MQCACF_UOW_START_TIME	3061	X'00000BF5'
MQCACF_UOW_LOG_START_DATE	3062	X'00000BF6'
MQCACF_UOW_LOG_START_TIME	3063	X'00000BF7'
MQCACF_UOW_LOG_EXTENT_NAME	3064	X'00000BF8'
MQCACF_PRINCIPAL_ENTITY_NAMES	3065	X'00000BF9'

Table 40. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCACF_GROUP_ENTITY_NAMES	3066	X'00000BFA'
MQCACF_AUTH_PROFILE_NAME	3067	X'00000BFB'
MQCACF_ENTITY_NAME	3068	X'00000BFC'
MQCACF_SERVICE_COMPONENT	3069	X'00000BFD'
MQCACF_RESPONSE_Q_MGR_NAME	3070	X'00000BFE'
MQCACF_CURRENT_LOG_EXTENT_NAME	3071	X'00000BFF'
MQCACF_RESTART_LOG_EXTENT_NAME	3072	X'00000C00'
MQCACF_MEDIA_LOG_EXTENT_NAME	3073	X'00000C01'
MQCACF_LOG_PATH	3074	X'00000C02'
MQCACF_COMMAND_MQSC	3075	X'00000C03'
MQCACF_Q_MGR_CPF	3076	X'00000C04'
MQCACF_USAGE_LOG_RBA	3078	X'00000C06'
MQCACF_USAGE_LOG_LRSN	3079	X'00000C07'
MQCACF_COMMAND_SCOPE	3080	X'00000C08'
MQCACF_ASID	3081	X'00000C09'
MQCACF_PSB_NAME	3082	X'00000C0A'
MQCACF_PST_ID	3083	X'00000C0B'
MQCACF_TASK_NUMBER	3084	X'00000C0C'
MQCACF_TRANSACTION_ID	3085	X'00000C0D'
MQCACF_Q_MGR_UOW_ID	3086	X'00000C0E'
MQCACF_ORIGIN_NAME	3088	X'00000C10'
MQCACF_ENV_INFO	3089	X'00000C11'
MQCACF_SECURITY_PROFILE	3090	X'00000C12'
MQCACF_CONFIGURATION_DATE	3091	X'00000C13'
MQCACF_CONFIGURATION_TIME	3092	X'00000C14'
MQCACF_FROM_CF_STRUC_NAME	3093	X'00000C15'
MQCACF_TO_CF_STRUC_NAME	3094	X'00000C16'
MQCACF_CF_STRUC_NAMES	3095	X'00000C17'
MQCACF_FAIL_DATE	3096	X'00000C18'
MQCACF_FAIL_TIME	3097	X'00000C19'
MQCACF_BACKUP_DATE	3098	X'00000C1A'
MQCACF_BACKUP_TIME	3099	X'00000C1B'
MQCACF_SYSTEM_NAME	3100	X'00000C1C'
MQCACF_CF_STRUC_BACKUP_START	3101	X'00000C1D'
MQCACF_CF_STRUC_BACKUP_END	3102	X'00000C1E'
MQCACF_CF_STRUC_LOG_Q_MGRS	3103	X'00000C1F'
MQCACF_FROM_STORAGE_CLASS	3104	X'00000C20'
MQCACF_TO_STORAGE_CLASS	3105	X'00000C21'
MQCACF_STORAGE_CLASS_NAMES	3106	X'00000C22'

Table 40. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCACF_DSG_NAME	3108	X'00000C24'
MQCACF_DB2_NAME	3109	X'00000C25'
MQCACF_SYSP_CMD_USER_ID	3110	X'00000C26'
MQCACF_SYSP_OTMA_GROUP	3111	X'00000C27'
MQCACF_SYSP_OTMA_MEMBER	3112	X'00000C28'
MQCACF_SYSP_OTMA_DRU_EXIT	3113	X'00000C29'
MQCACF_SYSP_OTMA_TPIPE_PFX	3114	X'00000C2A'
MQCACF_SYSP_ARCHIVE_PFX1	3115	X'00000C2B'
MQCACF_SYSP_ARCHIVE_UNIT1	3116	X'00000C2C'
MQCACF_SYSP_LOG_CORREL_ID	3117	X'00000C2D'
MQCACF_SYSP_UNIT_VOLSER	3118	X'00000C2E'
MQCACF_SYSP_Q_MGR_TIME	3119	X'00000C2F'
MQCACF_SYSP_Q_MGR_DATE	3120	X'00000C30'
MQCACF_SYSP_Q_MGR_RBA	3121	X'00000C31'
MQCACF_SYSP_LOG_RBA	3122	X'00000C32'
MQCACF_SYSP_SERVICE	3123	X'00000C33'
MQCACF_FROM_LISTENER_NAME	3124	X'00000C34'
MQCACF_TO_LISTENER_NAME	3125	X'00000C35'
MQCACF_FROM_SERVICE_NAME	3126	X'00000C36'
MQCACF_TO_SERVICE_NAME	3127	X'00000C37'
MQCACF_LAST_PUT_DATE	3128	X'00000C38'
MQCACF_LAST_PUT_TIME	3129	X'00000C39'
MQCACF_LAST_GET_DATE	3130	X'00000C3A'
MQCACF_LAST_GET_TIME	3131	X'00000C3B'
MQCACF_OPERATION_DATE	3132	X'00000C3C'
MQCACF_OPERATION_TIME	3133	X'00000C3D'
MQCACF_ACTIVITY_DESC	3134	X'00000C3E'
MQCACF_APPL_IDENTITY_DATA	3135	X'00000C3F'
MQCACF_APPL_ORIGIN_DATA	3136	X'00000C40'
MQCACF_PUT_DATE	3137	X'00000C41'
MQCACF_PUT_TIME	3138	X'00000C42'
MQCACF_REPLY_TO_Q	3139	X'00000C43'
MQCACF_REPLY_TO_Q_MGR	3140	X'00000C44'
MQCACF_RESOLVED_Q_NAME	3141	X'00000C45'
MQCACF_STRUC_ID	3142	X'00000C46'
MQCACF_VALUE_NAME	3143	X'00000C47'
MQCACF_SERVICE_START_DATE	3144	X'00000C48'
MQCACF_SERVICE_START_TIME	3145	X'00000C49'
MQCACF_SYSP_OFFLINE_RBA	3146	X'00000C4A'

Table 40. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCACF_SYSP_ARCHIVE_PFX2	3147	X'00000C4B'
MQCACF_SYSP_ARCHIVE_UNIT2	3148	X'00000C4C'
MQCACF_TO_TOPIC_NAME	3149	X'00000C4D'
MQCACF_FROM_TOPIC_NAME	3150	X'00000C4E'
MQCACF_TOPIC_NAMES	3151	X'00000C4F'
MQCACF_SUB_NAME	3152	X'00000C50'
MQCACF_DESTINATION_Q_MGR	3153	X'00000C51'
MQCACF_DESTINATION	3154	X'00000C52'
MQCACF_SUB_USER_ID	3156	X'00000C54'
MQCACF_SUB_USER_DATA	3159	X'00000C57'
MQCACF_SUB_SELECTOR	3160	X'00000C58'
MQCACF_LAST_PUB_DATE	3161	X'00000C59'
MQCACF_LAST_PUB_TIME	3162	X'00000C5A'
MQCACF_FROM_SUB_NAME	3163	X'00000C5B'
MQCACF_TO_SUB_NAME	3164	X'00000C5C'
MQCACF_LAST_MSG_TIME	3167	X'00000C5F'
MQCACF_LAST_MSG_DATE	3168	X'00000C60'
MQCACF_SUBSCRIPTION_POINT	3169	X'00000C61'
MQCACF_FILTER	3170	X'00000C62'
MQCACF_NONE	3171	X'00000C63'
MQCACF_ADMIN_TOPIC_NAMES	3172	X'00000C64'
MQCACF_LAST_USED	3172	X'00000C64'

MQCACH_* (Command format Character Channel Parameter Types)

Table 41. Values of constants		
Name	Decimal value	Hexadecimal value
MQCACH_FIRST	3501	X'00000DAD'
MQCACH_CHANNEL_NAME	3501	X'00000DAD'
MQCACH_DESC	3502	X'00000DAE'
MQCACH_MODE_NAME	3503	X'00000DAF'
MQCACH_TP_NAME	3504	X'00000DB0'
MQCACH_XMIT_Q_NAME	3505	X'00000DB1'
MQCACH_CONNECTION_NAME	3506	X'00000DB2'
MQCACH_MCA_NAME	3507	X'00000DB3'
MQCACH_SEC_EXIT_NAME	3508	X'00000DB4'
MQCACH_MSG_EXIT_NAME	3509	X'00000DB5'
MQCACH_SEND_EXIT_NAME	3510	X'00000DB6'
MQCACH_RCV_EXIT_NAME	3511	X'00000DB7'
MQCACH_CHANNEL_NAMES	3512	X'00000DB8'

Table 41. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCACH_SEC_EXIT_USER_DATA	3513	X'00000DB9'
MQCACH_MSG_EXIT_USER_DATA	3514	X'00000DBA'
MQCACH_SEND_EXIT_USER_DATA	3515	X'00000DBB'
MQCACH_RCV_EXIT_USER_DATA	3516	X'00000DBC'
MQCACH_USER_ID	3517	X'00000DBD'
MQCACH_PASSWORD	3518	X'00000DBE'
MQCACH_LOCAL_ADDRESS	3520	X'00000DC0'
MQCACH_LOCAL_NAME	3521	X'00000DC1'
MQCACH_LAST_MSG_TIME	3524	X'00000DC4'
MQCACH_LAST_MSG_DATE	3525	X'00000DC5'
MQCACH_MCA_USER_ID	3527	X'00000DC7'
MQCACH_CHANNEL_START_TIME	3528	X'00000DC8'
MQCACH_CHANNEL_START_DATE	3529	X'00000DC9'
MQCACH_MCA_JOB_NAME	3530	X'00000DCA'
MQCACH_LAST_LUWID	3531	X'00000DCB'
MQCACH_CURRENT_LUWID	3532	X'00000DCC'
MQCACH_FORMAT_NAME	3533	X'00000DCD'
MQCACH_MR_EXIT_NAME	3534	X'00000DCE'
MQCACH_MR_EXIT_USER_DATA	3535	X'00000DCF'
MQCACH_SSL_CIPHER_SPEC	3544	X'00000DD8'
MQCACH_SSL_PEER_NAME	3545	X'00000DD9'
MQCACH_SSL_HANDSHAKE_STAGE	3546	X'00000DDA'
MQCACH_SSL_SHORT_PEER_NAME	3547	X'00000ddb'
MQCACH_REMOTE_APPL_TAG	3548	X'00000DDC'
MQCACH_SSL_CERT_USER_ID	3549	X'00000DDD'
MQCACH_SSL_CERT_ISSUER_NAME	3550	X'00000DDE'
MQCACH_LU_NAME	3551	X'00000DDF'
MQCACH_IP_ADDRESS	3552	X'00000DE0'
MQCACH_TCP_NAME	3553	X'00000DE1'
MQCACH_LISTENER_NAME	3554	X'00000DE2'
MQCACH_LISTENER_DESC	3555	X'00000DE3'
MQCACH_LISTENER_START_DATE	3556	X'00000DE4'
MQCACH_LISTENER_START_TIME	3557	X'00000DE5'
MQCACH_SSL_KEY_RESET_DATE	3558	X'00000DE6'
MQCACH_SSL_KEY_RESET_TIME	3559	X'00000DE7'
MQCACH_LAST_USED	3559	X'00000DE7'

MQCADSD_* (CICS information header ADS Descriptors)

Table 42. Values of constants		
Name	Decimal value	Hexadecimal value
MQCADSD_NONE	0	X'00000000'
MQCADSD_SEND	1	X'00000001'
MQCADSD_RECV	16	X'00000010'
MQCADSD_MSGFORMAT	256	X'00000100'

MQCAFTY_* (Connection Affinity Values)

Table 43. Values of constants		
Name	Decimal value	Hexadecimal value
MQCAFTY_NONE	0	X'00000000'
MQCAFTY_PREFERRED	1	X'00000001'

MQCAMO_* (Command format Character Monitoring Parameter Types)

Table 44. Values of constants		
Name	Decimal value	Hexadecimal value
MQCAMO_FIRST	2701	X'00000A8D'
MQCAMO_CLOSE_DATE	2701	X'00000A8D'
MQCAMO_CLOSE_TIME	2702	X'00000A8E'
MQCAMO_CONN_DATE	2703	X'00000A8F'
MQCAMO_CONN_TIME	2704	X'00000A90'
MQCAMO_DISC_DATE	2705	X'00000A91'
MQCAMO_DISC_TIME	2706	X'00000A92'
MQCAMO_END_DATE	2707	X'00000A93'
MQCAMO_END_TIME	2708	X'00000A94'
MQCAMO_OPEN_DATE	2709	X'00000A95'
MQCAMO_OPEN_TIME	2710	X'00000A96'
MQCAMO_START_DATE	2711	X'00000A97'
MQCAMO_START_TIME	2712	X'00000A98'
MQCAMO_LAST_USED	2712	X'00000A98'

MQCBC_* (MQCBC constants structure)

Table 45. Structures of constants	
Name	Structure
MQCBC_STRUC_ID	"CBC¬"
MQCBC_STRUC_ID_ARRAY	'C', 'B', 'C', '¬'

Note: The symbol ¬ represents a single blank character.

Table 46. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBC_VERSION_1	1	X'00000001'

Table 46. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCBC_CURRENT_VERSION	1	X'00000001'

MQCBCF_* (MQCBC constants Flags)

Table 47. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBCF_NONE	0	X'00000000'
MQCBCF_READA_BUFFER_EMPTY	1	X'00000001'

MQCBCT_* (MQCBC constants Callback type)

Table 48. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBCT_START_CALL	1	X'00000001'
MQCBCT_STOP_CALL	2	X'00000002'
MQCBCT_REGISTER_CALL	3	X'00000003'
MQCBCT_DEREGISTER_CALL	4	X'00000004'
MQCBCT_EVENT_CALL	5	X'00000005'
MQCBCT_MSG_REMOVED	6	X'00000006'
MQCBCT_MSG_NOT_REMOVED	7	X'00000007'

MQCBD_* (MQCBD constants structure)

Table 49. Structures of constants	
Name	Structure
MQCBD_STRUC_ID	"CBD↵"
MQCBD_STRUC_ID_ARRAY	'C', 'B', 'D', '↵'

Note: The symbol ↵ represents a single blank character.

Table 50. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBD_VERSION_1	1	X'00000001'
MQCBD_CURRENT_VERSION	1	X'00000001'

MQCBDO_* (MQCBD constants Callback Options)

Table 51. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBDO_NONE	0	X'00000000'
MQCBDO_START_CALL	1	X'00000001'
MQCBDO_STOP_CALL	4	X'00000004'
MQCBDO_REGISTER_CALL	256	X'00000100'
MQCBDO_DEREGISTER_CALL	512	X'00000200'
MQCBDO_FAIL_IF QUIESCING	8192	X'00002000'

MQCBO_* (Create-Bag Options for mqCreateBag)

Table 52. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBO_NONE	0	X'00000000'
MQCBO_USER_BAG	0	X'00000000'
MQCBO_ADMIN_BAG	1	X'00000001'
MQCBO_COMMAND_BAG	16	X'00000010'
MQCBO_SYSTEM_BAG	32	X'00000020'
MQCBO_GROUP_BAG	64	X'00000040'
MQCBO_LIST_FORM_ALLOWED	2	X'00000002'
MQCBO_LIST_FORM_INHIBITED	0	X'00000000'
MQCBO_REORDER_AS_REQUIRED	4	X'00000004'
MQCBO_DO_NOT_REORDER	0	X'00000000'
MQCBO_CHECK_SELECTORS	8	X'00000008'
MQCBO_DO_NOT_CHECK_SELECTORS	0	X'00000000'

MQCBT_* (MQCBD constants This is the type of the Callback Function)

Table 53. Values of constants		
Name	Decimal value	Hexadecimal value
MQCBT_MESSAGE_CONSUMER	1	X'00000001'
MQCBT_EVENT_HANDLER	2	X'00000002'

MQCC_* (Completion Codes)

Table 54. Values of constants		
Name	Decimal value	Hexadecimal value
MQCC_OK	0	X'00000000'
MQCC_WARNING	1	X'00000001'
MQCC_FAILED	2	X'00000002'
MQCC_UNKNOWN	-1	X'FFFFFFFF'

MQCCSI_* (Coded Character Set Identifiers)

Table 55. Values of constants		
Name	Decimal value	Hexadecimal value
MQCCSI_UNDEFINED	0	X'00000000'
MQCCSI_DEFAULT	0	X'00000000'
MQCCSI_Q_MGR	0	X'00000000'
MQCCSI_INHERIT	-2	X'FFFFFFFE'
MQCCSI_EMBEDDED	-1	X'FFFFFFF'
MQCCSI_APPL	-3	X'FFFFFFFD'

MQCCT_* (CICS information header Conversational Task Options)

Table 56. Values of constants		
Name	Decimal value	Hexadecimal value
MQCCT_YES	1	X'00000001'
MQCCT_NO	0	X'00000000'

MQCD_* (Channel definition structure)

Table 57. Values of constants		
Name	Decimal value	Hexadecimal value
MQCD_VERSION_1	1	X'00000001'
MQCD_VERSION_2	2	X'00000002'
MQCD_VERSION_3	3	X'00000003'
MQCD_VERSION_4	4	X'00000004'
MQCD_VERSION_5	5	X'00000005'
MQCD_VERSION_6	6	X'00000006'
MQCD_VERSION_7	7	X'00000007'
MQCD_VERSION_8	8	X'00000008'
MQCD_VERSION_9	9	X'00000009'
MQCD_VERSION_10	10	X'0000000A'
MQCD_VERSION_11	11	X'0000000B'
MQCD_CURRENT_VERSION	11	X'0000000B'
MQCD_LENGTH_4	(value differs by platform or version)	
MQCD_LENGTH_5	(value differs by platform or version)	
MQCD_LENGTH_6	(value differs by platform or version)	
MQCD_LENGTH_7	(value differs by platform or version)	
MQCD_LENGTH_8	(value differs by platform or version)	
MQCD_LENGTH_9	(value differs by platform or version)	
MQCD_LENGTH_10	(value differs by platform or version)	
MQCD_LENGTH_11	(value differs by platform or version)	
MQCD_CURRENT_LENGTH	(value differs by platform or version)	

MQCDC_* (Channel Data Conversion)

Table 58. Values of constants		
Name	Decimal value	Hexadecimal value
MQCDC_SENDER_CONVERSION	1	X'00000001'
MQCDC_NO_SENDER_CONVERSION	0	X'00000000'

MQCERT_* (Certificate Validation Policy Type)

MQ_CERT_VAL_POLICY_DEFAULT	0	X'00000000'
MQ_CERT_VAL_POLICY_ANY	0	X'00000000'
MQ_CERT_VAL_POLICY_RFC5280	1	X'00000001'

MQCF_* (Capability Flags)

Table 59. Values of constants		
Name	Decimal value	Hexadecimal value
MQCF_NONE	0	X'00000000'
MQCF_DIST_LISTS	1	X'00000001'

MQCFAC_* (CICS information header Facility)

Table 60. Constant names and values	
Name	Hexadecimal value
MQCFAC_NONE	X'00...00' (8 nulls)
MQCFAC_NONE_ARRAY	'\0','\0',... (8 nulls)

MQCFBF_* (Command format byte string filter parameter structure)

Table 61. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFBF_STRUC_LENGTH_FIXED	20	X'00000014'

MQCFBS_* (Command format byte string parameter structure)

Table 62. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFBS_STRUC_LENGTH_FIXED	16	X'00000010'

MQCFC_* (Command format header Control Options)

Table 63. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFC_LAST	1	X'00000001'
MQCFC_NOT_LAST	0	X'00000000'

MQCFGR_* (Command format group parameter structure)

Table 64. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFGR_STRUC_LENGTH	16	X'00000010'

MQCFH_* (Command format header structure)

Table 65. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFH_STRUC_LENGTH	36	X'00000024'
MQCFH_VERSION_1	1	X'00000001'
MQCFH_VERSION_2	2	X'00000002'
MQCFH_VERSION_3	3	X'00000003'
MQCFH_CURRENT_VERSION	3	X'00000003'

MQCFIF_* (Command format integer filter parameter structure)

Table 66. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFIF_STRUC_LENGTH	20	X'00000014'

MQCFIL_* (Command format integer list parameter structure)

Table 67. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFIL_STRUC_LENGTH_FIXED	16	X'00000010'

MQCFIL64_* (Command format 64-bit integer list parameter structure)

Table 68. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFIL64_STRUC_LENGTH_FIXED	16	X'00000010'

MQCFIN_* (Command format integer parameter structure)

Table 69. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFIN_STRUC_LENGTH	16	X'00000010'

MQCFIN64_* (Command format 64-bit integer parameter structure)

Table 70. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFIN64_STRUC_LENGTH	24	X'00000018'

MQCFO_* (Command format Refresh Repository Options and Command format Remove Queues Options)

Command format Refresh Repository Options

Table 71. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFO_REFRESH_REPOSITORY_YES	1	X'00000001'
MQCFO_REFRESH_REPOSITORY_NO	0	X'00000000'

Command format Remove Queues Options

Table 72. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFO_REMOVE_QUEUES_YES	1	X'00000001'
MQCFO_REMOVE_QUEUES_NO	0	X'00000000'

MQCFOP_* (Command format Filter Operators)

Table 73. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFOP_LESS	1	X'00000001'
MQCFOP_EQUAL	2	X'00000002'
MQCFOP_GREATER	4	X'00000004'
MQCFOP_NOT_LESS	6	X'00000006'
MQCFOP_NOT_EQUAL	5	X'00000005'
MQCFOP_NOT_GREATER	3	X'00000003'
MQCFOP_LIKE	18	X'00000012'
MQCFOP_NOT_LIKE	21	X'00000015'
MQCFOP_CONTAINS	10	X'0000000A'
MQCFOP_EXCLUDES	13	X'0000000D'
MQCFOP_CONTAINS_GEN	26	X'0000001A'
MQCFOP_EXCLUDES_GEN	29	X'0000001D'

MQCFR_* (CF Recoverability)

Table 74. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFR_YES	1	X'00000001'
MQCFR_NO	0	X'00000000'

MQCFSF_* (Command format string filter parameter structure)

Table 75. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFSF_STRUC_LENGTH_FIXED	24	X'00000018'

MQCFSL_* (Command format string list parameter structure)

Table 76. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFSL_STRUC_LENGTH_FIXED	24	X'00000018'

MQCFST_* (Command format string parameter structure)

Table 77. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFST_STRUC_LENGTH_FIXED	20	X'00000014'

MQCFSTATUS_* (Command format CF Status)

Table 78. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFSTATUS_NOT_FOUND	0	X'00000000'
MQCFSTATUS_ACTIVE	1	X'00000001'
MQCFSTATUS_IN_RECOVER	2	X'00000002'
MQCFSTATUS_IN_BACKUP	3	X'00000003'
MQCFSTATUS_FAILED	4	X'00000004'
MQCFSTATUS_NONE	5	X'00000005'
MQCFSTATUS_UNKNOWN	6	X'00000006'
MQCFSTATUS_ADMIN_INCOMPLETE	20	X'00000014'
MQCFSTATUS_NEVER_USED	21	X'00000015'
MQCFSTATUS_NO_BACKUP	22	X'00000016'
MQCFSTATUS_NOT_FAILED	23	X'00000017'
MQCFSTATUS_NOT_RECOVERABLE	24	X'00000018'
MQCFSTATUS_XES_ERROR	25	X'00000019'

MQCFT_* (Command format Types of Structure)

Table 79. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFT_NONE	0	X'00000000'
MQCFT_COMMAND	1	X'00000001'
MQCFT_RESPONSE	2	X'00000002'
MQCFT_INTEGER	3	X'00000003'
MQCFT_STRING	4	X'00000004'
MQCFT_INTEGER_LIST	5	X'00000005'
MQCFT_STRING_LIST	6	X'00000006'
MQCFT_EVENT	7	X'00000007'
MQCFT_USER	8	X'00000008'
MQCFT_BYTE_STRING	9	X'00000009'
MQCFT_TRACE_ROUTE	10	X'0000000A'
MQCFT_REPORT	12	X'0000000C'
MQCFT_INTEGER_FILTER	13	X'0000000D'
MQCFT_STRING_FILTER	14	X'0000000E'
MQCFT_BYTE_STRING_FILTER	15	X'0000000F'
MQCFT_COMMAND_XR	16	X'00000010'

Table 79. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCFT_XR_MSG	17	X'00000011'
MQCFT_XR_ITEM	18	X'00000012'
MQCFT_XR_SUMMARY	19	X'00000013'
MQCFT_GROUP	20	X'00000014'
MQCFT_STATISTICS	21	X'00000015'
MQCFT_ACCOUNTING	22	X'00000016'
MQCFT_INTEGER64	23	X'00000017'
MQCFT_INTEGER64_LIST	25	X'00000019'

MQCFTYPE_* (Command format CF Types)

Table 80. Values of constants		
Name	Decimal value	Hexadecimal value
MQCFTYPE_APPL	0	X'00000000'
MQCFTYPE_ADMIN	1	X'00000001'

MQCFUNC_* (CICS information header Functions)

Table 81. Structures of constants	
Name	Structure
MQCFUNC_MQCONN	"CONN"
MQCFUNC_MQGET	"GET¬"
MQCFUNC_MQINQ	"INQ¬"
MQCFUNC_MQOPEN	"OPEN"
MQCFUNC_MQPUT	"PUT¬"
MQCFUNC_MQPUT1	"PUT1"
MQCFUNC_NONE	"¬¬¬¬"
MQCFUNC_MQCONN_ARRAY	'C','O','N','N'
MQCFUNC_MQGET_ARRAY	'G','E','T','¬'
MQCFUNC_MQINQ_ARRAY	'I','N','Q','¬'
MQCFUNC_MQOPEN_ARRAY	'O','P','E','N'
MQCFUNC_MQPUT_ARRAY	'P','U','T','¬'
MQCFUNC_MQPUT1_ARRAY	'P','U','T','1'
MQCFUNC_NONE_ARRAY	'¬','¬','¬','¬'

Note: The symbol ¬ represents a single blank character.

MQCGWI_* (CICS information header Get Wait Interval)

Table 82. Values of constants		
Name	Decimal value	Hexadecimal value
MQCGWI_DEFAULT	-2	X'FFFFFFFFE'

MQCHAD_* (Channel Auto Definition)

Table 83. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHAD_DISABLED	0	X'00000000'
MQCHAD_ENABLED	1	X'00000001'

MQCHIDS_* (Command format Indoubt Status)

Table 84. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHIDS_NOT_INDOUBT	0	X'00000000'
MQCHIDS_INDOUBT	1	X'00000001'

MQCHLD_* (Command format Channel Dispositions)

Table 85. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHLD_ALL	-1	X'FFFFFFFF'
MQCHLD_DEFAULT	1	X'00000001'
MQCHLD_SHARED	2	X'00000002'
MQCHLD_PRIVATE	4	X'00000004'
MQCHLD_FIXSHARED	5	X'00000005'

MQCHS_* (Command format Channel Status)

Table 86. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHS_INACTIVE	0	X'00000000'
MQCHS_BINDING	1	X'00000001'
MQCHS_STARTING	2	X'00000002'
MQCHS_RUNNING	3	X'00000003'
MQCHS_STOPPING	4	X'00000004'
MQCHS_RETRYING	5	X'00000005'
MQCHS_STOPPED	6	X'00000006'
MQCHS_REQUESTING	7	X'00000007'
MQCHS_PAUSED	8	X'00000008'
MQCHS_INITIALIZING	13	X'0000000D'
MQCHS_SWITCHING	14	X'0000000E'

MQCHSH_* (Command format Channel Shared Restart Options)

Table 87. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHSH_RESTART_NO	0	X'00000000'
MQCHSH_RESTART_YES	1	X'00000001'

MQCHSR_* (Command format Channel Stop Options)

Table 88. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHSR_STOP_NOT_REQUESTED	0	X'00000000'
MQCHSR_STOP_REQUESTED	1	X'00000001'

MQCHSSTATE_* (Command format Channel Substates)

Table 89. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHSSTATE_OTHER	0	X'00000000'
MQCHSSTATE_END_OF_BATCH	100	X'00000064'
MQCHSSTATE_SENDING	200	X'000000C8'
MQCHSSTATE_RECEIVING	300	X'0000012C'
MQCHSSTATE_SERIALIZING	400	X'00000190'
MQCHSSTATE_RESYNCHING	500	X'000001F4'
MQCHSSTATE_HEARTBEATING	600	X'00000258'
MQCHSSTATE_IN_SCYEXIT	700	X'000002BC'
MQCHSSTATE_IN_RCVEXIT	800	X'00000320'
MQCHSSTATE_IN_SENDEXIT	900	X'00000384'
MQCHSSTATE_IN_MSGEXIT	1000	X'000003E8'
MQCHSSTATE_IN_MREXIT	1100	X'0000044C'
MQCHSSTATE_IN_CHADEXIT	1200	X'000004B0'
MQCHSSTATE_NET_CONNECTING	1250	X'000004E2'
MQCHSSTATE_SSL_HANDSHAKING	1300	X'00000514'
MQCHSSTATE_NAME_SERVER	1400	X'00000578'
MQCHSSTATE_IN_MQPUT	1500	X'000005DC'
MQCHSSTATE_IN_MQGET	1600	X'00000640'
MQCHSSTATE_IN_MQI_CALL	1700	X'000006A4'
MQCHSSTATE_COMPRESSING	1800	X'00000708'

MQCHT_* (Channel Types)

Table 90. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHT_SENDER	1	X'00000001'
MQCHT_SERVER	2	X'00000002'
MQCHT_RECEIVER	3	X'00000003'
MQCHT_REQUESTER	4	X'00000004'
MQCHT_ALL	5	X'00000005'
MQCHT_CLNTCONN	6	X'00000006'
MQCHT_SVRCONN	7	X'00000007'
MQCHT_CLUSRCVR	8	X'00000008'

Table 90. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCHT_CLUSSDR	9	X'00000009'

MQCHTAB_* (Command format Channel Table Types)

Table 91. Values of constants		
Name	Decimal value	Hexadecimal value
MQCHTAB_Q_MGR	1	X'00000001'
MQCHTAB_CLNTCONN	2	X'00000002'

MQCI_* (Correlation Identifier)

Table 92. Constant names and values	
Name	Value
MQCI_NONE	X'00...00' (24 nulls)
MQCI_NONE_ARRAY	'\0', '\0', ... (24 nulls)
MQCI_NEW_SESSION	X'414D5121...'
MQCI_NEW_SESSION_ARRAY	'\x41', '\x4D', '\51', '\x21', ...

MQCIH_* (CICS information header structure and Flags)

CICS information header structure

Table 93. Structures of constants	
Name	Structure
MQCIH_STRUC_ID	"CIH¬"
MQCIH_STRUC_ID_ARRAY	'C', 'I', 'H', '¬'

Note: The symbol ¬ represents a single blank character.

Table 94. Values of constants		
Name	Decimal value	Hexadecimal value
MQCIH_VERSION_1	1	X'00000001'
MQCIH_VERSION_2	2	X'00000002'
MQCIH_CURRENT_VERSION	2	X'00000002'
MQCIH_LENGTH_1	164	X'000000A4'
MQCIH_LENGTH_2	180	X'000000B4'
MQCIH_CURRENT_LENGTH	180	X'000000B4'

CICS information header Flags

Table 95. Values of constants		
Name	Decimal value	Hexadecimal value
MQCIH_NONE	0	X'00000000'
MQCIH_PASS_EXPIRATION	1	X'00000001'
MQCIH_UNLIMITED_EXPIRATION	0	X'00000000'

Table 95. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCIH_REPLY_WITHOUT_NULLS	2	X'00000002'
MQCIH_REPLY_WITH_NULLS	0	X'00000000'
MQCIH_SYNC_ON_RETURN	4	X'00000004'
MQCIH_NO_SYNC_ON_RETURN	0	X'00000000'

MQCLCT_* (Cluster Cache Types)

Table 96. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLCT_STATIC	0	X'00000000'
MQCLCT_DYNAMIC	1	X'00000001'

MQCLRS_* (Command format Clear Topic String Scope)

Table 97. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLRS_LOCAL	1	X'00000001'
MQCLRS_GLOBAL	2	X'00000002'

MQCLRT_* (Command format Clear Topic String Type)

Table 98. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLRT_RETAINED	1	X'00000001'

MQCLT_* (CICS information header Link Types)

Table 99. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLT_PROGRAM	1	X'00000001'
MQCLT_TRANSACTION	2	X'00000002'

MQCLWL_* (Cluster Workload)

Table 100. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLWL_USEQ_LOCAL	0	X'00000000'
MQCLWL_USEQ_ANY	1	X'00000001'
MQCLWL_USEQ_AS_Q_MGR	-3	X'FFFFFFFD'

MQCLXQ_* (Cluster transmission queue type)

MQCLXQ_* are the values you can set in the DEFCLXQ queue manager attribute. The DEFCLXQ attribute controls which transmission queue is selected by default by cluster-sender channels to get messages from, to send the messages to cluster-receiver channels.

Table 101. Values of constants		
Name	Decimal value	Hexadecimal value
MQCLXQ_SCTQ	0	X'00000000'
MQCLXQ_CHANNEL	1	X'00000001'

Related reference

[“DefClusterXmitQueueType \(MQLONG\)” on page 783](#)

The DefClusterXmitQueueType attribute controls which transmission queue is selected by default by cluster-sender channels to get messages from, to send the messages to cluster-receiver channels.

[Change Queue Manager](#)

[Inquire Queue Manager](#)

[Inquire Queue Manager \(Response\)](#)

[“MQINQ - Inquire object attributes” on page 677](#)

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object.

MQCMD_* (Command Codes)

Table 102. Values of constants		
Name	Decimal value	Hexadecimal value
MQCMD_NONE	0	X'00000000'
MQCMD_CHANGE_Q_MGR	1	X'00000001'
MQCMD_INQUIRE_Q_MGR	2	X'00000002'
MQCMD_CHANGE_PROCESS	3	X'00000003'
MQCMD_COPY_PROCESS	4	X'00000004'
MQCMD_CREATE_PROCESS	5	X'00000005'
MQCMD_DELETE_PROCESS	6	X'00000006'
MQCMD_INQUIRE_PROCESS	7	X'00000007'
MQCMD_CHANGE_Q	8	X'00000008'
MQCMD_CLEAR_Q	9	X'00000009'
MQCMD_COPY_Q	10	X'0000000A'
MQCMD_CREATE_Q	11	X'0000000B'
MQCMD_DELETE_Q	12	X'0000000C'
MQCMD_INQUIRE_Q	13	X'0000000D'
MQCMD_REFRESH_Q_MGR	16	X'00000010'
MQCMD_RESET_Q_STATS	17	X'00000011'
MQCMD_INQUIRE_Q_NAMES	18	X'00000012'
MQCMD_INQUIRE_PROCESS_NAMES	19	X'00000013'
MQCMD_INQUIRE_CHANNEL_NAMES	20	X'00000014'
MQCMD_CHANGE_CHANNEL	21	X'00000015'
MQCMD_COPY_CHANNEL	22	X'00000016'

Table 102. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCMD_CREATE_CHANNEL	23	X'00000017'
MQCMD_DELETE_CHANNEL	24	X'00000018'
MQCMD_INQUIRE_CHANNEL	25	X'00000019'
MQCMD_PING_CHANNEL	26	X'0000001A'
MQCMD_RESET_CHANNEL	27	X'0000001B'
MQCMD_START_CHANNEL	28	X'0000001C'
MQCMD_STOP_CHANNEL	29	X'0000001D'
MQCMD_START_CHANNEL_INIT	30	X'0000001E'
MQCMD_START_CHANNEL_LISTENER	31	X'0000001F'
MQCMD_CHANGE_NAMELIST	32	X'00000020'
MQCMD_COPY_NAMELIST	33	X'00000021'
MQCMD_CREATE_NAMELIST	34	X'00000022'
MQCMD_DELETE_NAMELIST	35	X'00000023'
MQCMD_INQUIRE_NAMELIST	36	X'00000024'
MQCMD_INQUIRE_NAMELIST_NAMES	37	X'00000025'
MQCMD_ESCAPE	38	X'00000026'
MQCMD_RESOLVE_CHANNEL	39	X'00000027'
MQCMD_PING_Q_MGR	40	X'00000028'
MQCMD_INQUIRE_Q_STATUS	41	X'00000029'
MQCMD_INQUIRE_CHANNEL_STATUS	42	X'0000002A'
MQCMD_CONFIG_EVENT	43	X'0000002B'
MQCMD_Q_MGR_EVENT	44	X'0000002C'
MQCMD_PERFM_EVENT	45	X'0000002D'
MQCMD_CHANNEL_EVENT	46	X'0000002E'
MQCMD_DELETE_PUBLICATION	60	X'0000003C'
MQCMD_DEREGISTER_PUBLISHER	61	X'0000003D'
MQCMD_DEREGISTER_SUBSCRIBER	62	X'0000003E'
MQCMD_PUBLISH	63	X'0000003F'
MQCMD_REGISTER_PUBLISHER	64	X'00000040'
MQCMD_REGISTER_SUBSCRIBER	65	X'00000041'
MQCMD_REQUEST_UPDATE	66	X'00000042'
MQCMD_BROKER_INTERNAL	67	X'00000043'
MQCMD_ACTIVITY_MSG	69	X'00000045'
MQCMD_INQUIRE_CLUSTER_Q_MGR	70	X'00000046'
MQCMD_RESUME_Q_MGR_CLUSTER	71	X'00000047'
MQCMD_SUSPEND_Q_MGR_CLUSTER	72	X'00000048'
MQCMD_REFRESH_CLUSTER	73	X'00000049'
MQCMD_RESET_CLUSTER	74	X'0000004A'
MQCMD_TRACE_ROUTE	75	X'0000004B'

Table 102. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCMD_REFRESH_SECURITY	78	X'0000004E'
MQCMD_CHANGE_AUTH_INFO	79	X'0000004F'
MQCMD_COPY_AUTH_INFO	80	X'00000050'
MQCMD_CREATE_AUTH_INFO	81	X'00000051'
MQCMD_DELETE_AUTH_INFO	82	X'00000052'
MQCMD_INQUIRE_AUTH_INFO	83	X'00000053'
MQCMD_INQUIRE_AUTH_INFO_NAMES	84	X'00000054'
MQCMD_INQUIRE_CONNECTION	85	X'00000055'
MQCMD_STOP_CONNECTION	86	X'00000056'
MQCMD_INQUIRE_AUTH_RECS	87	X'00000057'
MQCMD_INQUIRE_ENTITY_AUTH	88	X'00000058'
MQCMD_DELETE_AUTH_REC	89	X'00000059'
MQCMD_SET_AUTH_REC	90	X'0000005A'
MQCMD_LOGGER_EVENT	91	X'0000005B'
MQCMD_RESET_Q_MGR	92	X'0000005C'
MQCMD_CHANGE_LISTENER	93	X'0000005D'
MQCMD_COPY_LISTENER	94	X'0000005E'
MQCMD_CREATE_LISTENER	95	X'0000005F'
MQCMD_DELETE_LISTENER	96	X'00000060'
MQCMD_INQUIRE_LISTENER	97	X'00000061'
MQCMD_INQUIRE_LISTENER_STATUS	98	X'00000062'
MQCMD_COMMAND_EVENT	99	X'00000063'
MQCMD_CHANGE_SECURITY	100	X'00000064'
MQCMD_CHANGE_CF_STRUC	101	X'00000065'
MQCMD_CHANGE_STG_CLASS	102	X'00000066'
MQCMD_CHANGE_TRACE	103	X'00000067'
MQCMD_ARCHIVE_LOG	104	X'00000068'
MQCMD_BACKUP_CF_STRUC	105	X'00000069'
MQCMD_CREATE_BUFFER_POOL	106	X'0000006A'
MQCMD_CREATE_PAGE_SET	107	X'0000006B'
MQCMD_CREATE_CF_STRUC	108	X'0000006C'
MQCMD_CREATE_STG_CLASS	109	X'0000006D'
MQCMD_COPY_CF_STRUC	110	X'0000006E'
MQCMD_COPY_STG_CLASS	111	X'0000006F'
MQCMD_DELETE_CF_STRUC	112	X'00000070'
MQCMD_DELETE_STG_CLASS	113	X'00000071'
MQCMD_INQUIRE_ARCHIVE	114	X'00000072'
MQCMD_INQUIRE_CF_STRUC	115	X'00000073'
MQCMD_INQUIRE_CF_STRUC_STATUS	116	X'00000074'

Table 102. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCMD_INQUIRE_CMD_SERVER	117	X'00000075'
MQCMD_INQUIRE_CHANNEL_INIT	118	X'00000076'
MQCMD_INQUIRE_QSG	119	X'00000077'
MQCMD_INQUIRE_LOG	120	X'00000078'
MQCMD_INQUIRE_SECURITY	121	X'00000079'
MQCMD_INQUIRE_STG_CLASS	122	X'0000007A'
MQCMD_INQUIRE_SYSTEM	123	X'0000007B'
MQCMD_INQUIRE_THREAD	124	X'0000007C'
MQCMD_INQUIRE_TRACE	125	X'0000007D'
MQCMD_INQUIRE_USAGE	126	X'0000007E'
MQCMD_MOVE_Q	127	X'0000007F'
MQCMD_RECOVER_BSDS	128	X'00000080'
MQCMD_RECOVER_CF_STRUC	129	X'00000081'
MQCMD_RESET_TPIPE	130	X'00000082'
MQCMD_RESOLVE_INDOUBT	131	X'00000083'
MQCMD_RESUME_Q_MGR	132	X'00000084'
MQCMD_REVERIFY_SECURITY	133	X'00000085'
MQCMD_SET_ARCHIVE	134	X'00000086'
MQCMD_SET_LOG	136	X'00000088'
MQCMD_SET_SYSTEM	137	X'00000089'
MQCMD_START_CMD_SERVER	138	X'0000008A'
MQCMD_START_Q_MGR	139	X'0000008B'
MQCMD_START_TRACE	140	X'0000008C'
MQCMD_STOP_CHANNEL_INIT	141	X'0000008D'
MQCMD_STOP_CHANNEL_LISTENER	142	X'0000008E'
MQCMD_STOP_CMD_SERVER	143	X'0000008F'
MQCMD_STOP_Q_MGR	144	X'00000090'
MQCMD_STOP_TRACE	145	X'00000091'
MQCMD_SUSPEND_Q_MGR	146	X'00000092'
MQCMD_INQUIRE_CF_STRUC_NAMES	147	X'00000093'
MQCMD_INQUIRE_STG_CLASS_NAMES	148	X'00000094'
MQCMD_CHANGE_SERVICE	149	X'00000095'
MQCMD_COPY_SERVICE	150	X'00000096'
MQCMD_CREATE_SERVICE	151	X'00000097'
MQCMD_DELETE_SERVICE	152	X'00000098'
MQCMD_INQUIRE_SERVICE	153	X'00000099'
MQCMD_INQUIRE_SERVICE_STATUS	154	X'0000009A'
MQCMD_START_SERVICE	155	X'0000009B'
MQCMD_STOP_SERVICE	156	X'0000009C'





Table 102. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCMD_DELETE_BUFFER_POOL	157	X'0000009D'
MQCMD_DELETE_PAGE_SET	158	X'0000009E'
MQCMD_CHANGE_BUFFER_POOL	159	X'0000009F'
MQCMD_CHANGE_PAGE_SET	160	X'000000A0'
MQCMD_INQUIRE_Q_MGR_STATUS	161	X'000000A1'
MQCMD_CREATE_LOG	162	X'000000A2'
MQCMD_STATISTICS_MQI	164	X'000000A4'
MQCMD_STATISTICS_Q	165	X'000000A5'
MQCMD_STATISTICS_CHANNEL	166	X'000000A6'
MQCMD_ACCOUNTING_MQI	167	X'000000A7'
MQCMD_ACCOUNTING_Q	168	X'000000A8'
MQCMD_INQUIRE_AUTH_SERVICE	169	X'000000A9'
MQCMD_CHANGE_TOPIC	170	X'000000AA'
MQCMD_COPY_TOPIC	171	X'000000AB'
MQCMD_CREATE_TOPIC	172	X'000000AC'
MQCMD_DELETE_TOPIC	173	X'000000AD'
MQCMD_INQUIRE_TOPIC	174	X'000000AE'
MQCMD_INQUIRE_TOPIC_NAMES	175	X'000000AF'
MQCMD_INQUIRE_SUBSCRIPTION	176	X'000000B0'
MQCMD_CREATE_SUBSCRIPTION	177	X'000000B1'
MQCMD_CHANGE_SUBSCRIPTION	178	X'000000B2'
MQCMD_DELETE_SUBSCRIPTION	179	X'000000B3'
MQCMD_COPY_SUBSCRIPTION	181	X'000000B5'
MQCMD_INQUIRE_SUB_STATUS	182	X'000000B6'
MQCMD_INQUIRE_TOPIC_STATUS	183	X'000000B7'
MQCMD_CLEAR_TOPIC_STRING	184	X'000000B8'
MQCMD_INQUIRE_PUBSUB_STATUS	185	X'000000B9'
MQCMD_PURGE_CHANNEL	195	X'000000C3'

MQCMDI_* (Command format Command Information Values)


Table 103. Values of constants		
Name	Decimal value	Hexadecimal value
MQCMDI_CMDSCOPE_ACCEPTED	1	X'00000001'
MQCMDI_CMDSCOPE_GENERATED	2	X'00000002'
MQCMDI_CMDSCOPE_COMPLETED	3	X'00000003'
MQCMDI_QSG_DISP_COMPLETED	4	X'00000004'
MQCMDI_COMMAND_ACCEPTED	5	X'00000005'
MQCMDI_CLUSTER_REQUEST_QUEUED	6	X'00000006'
MQCMDI_CHANNEL_INIT_STARTED	7	X'00000007'

Table 103. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCMDI_RECOVER_STARTED	11	X'0000000B'
MQCMDI_BACKUP_STARTED	12	X'0000000C'
MQCMDI_RECOVER_COMPLETED	13	X'0000000D'
MQCMDI_SEC_TIMER_ZERO	14	X'0000000E'
MQCMDI_REFRESH_CONFIGURATION	16	X'00000010'
MQCMDI_SEC_SIGNOFF_ERROR	17	X'00000011'
MQCMDI_IMS_BRIDGE_SUSPENDED	18	X'00000012'
MQCMDI_DB2_SUSPENDED	19	X'00000013'
MQCMDI_DB2_OBSOLETE_MSGS	20	X'00000014'
MQCMDI_SEC_UPPERCASE	21	X'00000015'
MQCMDI_SEC_MIXEDCASE	22	X'00000016'

MQCMDL_* (Command Levels)

Table 104. Constant names and values	
Name	Value
MQCMDL_LEVEL_600	600
MQCMDL_LEVEL_700	700
MQCMDL_LEVEL_701	701
MQCMDL_LEVEL_710	710
MQCMDL_LEVEL_711	711
MQCMDL_LEVEL_750	750
MQCMDL_LEVEL_800	800
  MQCMDL_LEVEL_801	801
  MQCMDL_LEVEL_802	802

Notes:

1. MQCMDL_LEVEL_801 applies only to UNIX platforms and IBM i, when you install IBM MQ 8.0.0, Fix Pack 2.
2.  MQCMDL_LEVEL_802 applies only to Windows, UNIX platforms, and IBM i, when you install IBM MQ Version 8.0.0, Fix Pack 3

MQCMHO_* (Create message handle options and structure)

Create message handle options structure

Table 105. Structures of constants	
Name	Structure
MQCMHO_STRUC_ID	"CMHO"
MQCMHO_STRUC_ID_ARRAY	'C', 'M', 'H', 'O'

Note: The symbol ¬ represents a single blank character.

Table 106. Values of constants		
Name	Decimal value	Hexadecimal value
MQCMHO_VERSION_1	1	X'00000001'
MQCMHO_CURRENT_VERSION	1	X'00000001'

Create Message Handle Options

Table 107. Values of constants		
Name	Decimal value	Hexadecimal value
MQCMHO_DEFAULT_VALIDATION	0	X'00000000'
MQCMHO_NO_VALIDATION	1	X'00000001'
MQCMHO_VALIDATE	2	X'00000002'
MQCMHO_NONE	0	X'00000000'

MQCNO_* (Connect options and structure)

Connect options structure

Table 108. Structures of constants	
Name	Structure
MQCNO_STRUC_ID	"CNO¬"
MQCNO_STRUC_ID_ARRAY	'C', 'N', 'O', '¬'

Note: The symbol ¬ represents a single blank character.

Table 109. Values of constants		
Name	Decimal value	Hexadecimal value
MQCNO_VERSION_1	1	X'00000001'
MQCNO_VERSION_2	2	X'00000002'
MQCNO_VERSION_3	3	X'00000003'
MQCNO_VERSION_4	4	X'00000004'
MQCNO_VERSION_5	5	X'00000005'
MQCNO_CURRENT_VERSION	5	X'00000005'

Connect Options

Table 110. Values of constants		
Name	Decimal value	Hexadecimal value
MQCNO_STANDARD_BINDING	0	X'00000000'
MQCNO_FASTPATH_BINDING	1	X'00000001'
MQCNO_SERIALIZE_CONN_TAG_Q_MGR	2	X'00000002'
MQCNO_SERIALIZE_CONN_TAG_QSG	4	X'00000004'
MQCNO_RESTRICT_CONN_TAG_Q_MGR	8	X'00000008'
MQCNO_RESTRICT_CONN_TAG_QSG	16	X'00000010'
MQCNO_HANDLE_SHARE_NONE	32	X'00000020'
MQCNO_HANDLE_SHARE_BLOCK	64	X'00000040'

Table 110. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCNO_HANDLE_SHARE_NO_BLOCK	128	X'00000080'
MQCNO_SHARED_BINDING	256	X'00000100'
MQCNO_ISOLATED_BINDING	512	X'00000200'
MQCNO_LOCAL_BINDING	1024	X'00000400'
MQCNO_CLIENT_BINDING	2048	X'00000800'
MQCNO_ACCOUNTING_MQI_ENABLED	4096	X'00001000'
MQCNO_ACCOUNTING_MQI_DISABLED	8192	X'00002000'
MQCNO_ACCOUNTING_Q_ENABLED	16384	X'00004000'
MQCNO_ACCOUNTING_Q_DISABLED	32768	X'00008000'
MQCNO_NO_CONV_SHARING	65536	X'00010000'
MQCNO_ALL_CONVS_SHARE	262144	X'00040000'
MQCNO_CD_FOR_OUTPUT_ONLY	524288	X'00080000'
MQCNO_USE_CD_SELECTION	1048576	X'00100000'
MQCNO_RECONNECT	16777216	X'01000000'
MQCNO_RECONNECT_AS_DEF	0	X'00000000'
MQCNO_RECONNECT_DISABLED	33554432	X'02000000'
MQCNO_RECONNECT_Q_MGR	67108864	X'04000000'
MQCNO_ACTIVITY_TRACE_ENABLED	134217728	X'08000000'
MQCNO_ACTIVITY_TRACE_DISABLED	268435456	X'10000000'
MQCNO_NONE	0	X'00000000'

MQCO_* (Close Options)

Table 111. Values of constants		
Name	Decimal value	Hexadecimal value
MQCO_IMMEDIATE	0	X'00000000'
MQCO_NONE	0	X'00000000'
MQCO_DELETE	1	X'00000001'
MQCO_DELETE_PURGE	2	X'00000002'
MQCO_KEEP_SUB	4	X'00000004'
MQCO_REMOVE_SUB	8	X'00000008'
MQCO_QUIESCE	32	X'00000020'

MQCODL_* (CICS information header Output Data Length)

Table 112. Values of constants		
Name	Decimal value	Hexadecimal value
MQCODL_AS_INPUT	-1	X'FFFFFFFF'

MQCOMPRESS_* (Channel Compression)

Table 113. Values of constants		
Name	Decimal value	Hexadecimal value
MQCOMPRESS_NOT_AVAILABLE	-1	X'FFFFFFFF'
MQCOMPRESS_NONE	0	X'00000000'
MQCOMPRESS_RLE	1	X'00000001'
MQCOMPRESS_ZLIBFAST	2	X'00000002'
MQCOMPRESS_ZLIBHIGH	4	X'00000004'
MQCOMPRESS_SYSTEM	8	X'00000008'
MQCOMPRESS_ANY	268435455	X'0FFFFFFF'

MQCONNID_* (Connection Identifier)

Table 114. Constant names and values	
Name	Value
MQCONNID_NONE	X'00...00' (24 nulls)
MQCONNID_NONE_ARRAY	'\0','\0',... (24 nulls)

MQCOPY_* (Property Copy Options)

Table 115. Values of constants		
Name	Decimal value	Hexadecimal value
MQCOPY_NONE	0	X'00000000'
MQCOPY_ALL	1	X'00000001'
MQCOPY_FORWARD	2	X'00000002'
MQCOPY_PUBLISH	4	X'00000004'
MQCOPY_REPLY	8	X'00000008'
MQCOPY_REPORT	16	X'00000010'
MQCOPY_DEFAULT	22	X'00000016'

MQCQT_* (Cluster Queue Types)

Table 116. Values of constants		
Name	Decimal value	Hexadecimal value
MQCQT_LOCAL_Q	1	X'00000001'
MQCQT_ALIAS_Q	2	X'00000002'
MQCQT_REMOTE_Q	3	X'00000003'
MQCQT_Q_MGR_ALIAS	4	X'00000004'

MQCRC_* (CICS information header Return Codes)

Table 117. Values of constants		
Name	Decimal value	Hexadecimal value
MQCRC_OK	0	X'00000000'
MQCRC_CICS_EXEC_ERROR	1	X'00000001'

Table 117. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQCRC_MQ_API_ERROR	2	X'00000002'
MQCRC_BRIDGE_ERROR	3	X'00000003'
MQCRC_BRIDGE_ABEND	4	X'00000004'
MQCRC_APPLICATION_ABEND	5	X'00000005'
MQCRC_SECURITY_ERROR	6	X'00000006'
MQCRC_PROGRAM_NOT_AVAILABLE	7	X'00000007'
MQCRC_BRIDGE_TIMEOUT	8	X'00000008'
MQCRC_TRANSID_NOT_AVAILABLE	9	X'00000009'

MQCS_* (MQCBC constants Consumer state)

Table 118. Values of constants		
Name	Decimal value	Hexadecimal value
MQCS_NONE	0	X'00000000'
MQCS_SUSPENDED_TEMPORARY	1	X'00000001'
MQCS_SUSPENDED_USER_ACTION	2	X'00000002'
MQCS_SUSPENDED	3	X'00000003'
MQCS_STOPPED	4	X'00000004'

MQCSC_* (CICS information header Start Codes)

Table 119. Structures of constants	
Name	Structure
MQCSC_START	"S¬¬"
MQCSC_STARTDATA	"SD¬¬"
MQCSC_TERMINPUT	"TD¬¬"
MQCSC_NONE	"¬¬¬¬"
MQCSC_START_ARRAY	'S','¬','¬','¬'
MQCSC_STARTDATA_ARRAY	'S','D','¬','¬'
MQCSC_TERMINPUT_ARRAY	'T','D','¬','¬'
MQCSC_NONE_ARRAY	'¬','¬','¬','¬'

Note: The symbol ¬ represents a single blank character.

MQCSP_* (Connection security parameters structure and Authentication Types)

Connection security parameters structure

Table 120. Structures of constants	
Name	Structure
MQCSP_STRUC_ID	"CSP¬"
MQCSP_STRUC_ID_ARRAY	'C','S','P','¬'

Note: The symbol – represents a single blank character.

Table 121. Values of constants		
Name	Decimal value	Hexadecimal value
MQCSP_VERSION_1	1	X'00000001'
MQCSP_CURRENT_VERSION	1	X'00000001'

Connection security parameters Authentication Types

Table 122. Values of constants		
Name	Decimal value	Hexadecimal value
MQCSP_AUTH_NONE	0	X'00000000'
MQCSP_AUTH_USER_ID_AND_PWD	1	X'00000001'

MQCSRV_* (Command Server Options)

Table 123. Values of constants		
Name	Decimal value	Hexadecimal value
MQCSRV_CONVERT_NO	0	X'00000000'
MQCSRV_CONVERT_YES	1	X'00000001'
MQCSRV_DLQ_NO	0	X'00000000'
MQCSRV_DLQ_YES	1	X'00000001'

MQCT_* (Queue Manager Connection Tag)

Table 124. Constant names and values	
Name	Value
MQCT_NONE	X'00...00' (128 nulls)
MQCT_NONE_ARRAY	'\0', '\0', ... (128 nulls)

MQCTES_* (CICS information header Task End Status)

Table 125. Values of constants		
Name	Decimal value	Hexadecimal value
MQCTES_NOSYNC	0	X'00000000'
MQCTES_COMMIT	256	X'00000100'
MQCTES_BACKOUT	4352	X'00001100'
MQCTES_ENDTASK	65536	X'00010000'

MQCTLO_* (MQCTL options structure and Consumer Control Options)

MQCTL options structure

Table 126. Structures of constants	
Name	Structure
MQCTLO_STRUC_ID	"CTLO"
MQCTLO_STRUC_ID_ARRAY	'C', 'T', 'L', 'O'

Note: The symbol ¬ represents a single blank character.

Table 127. Values of constants		
Name	Decimal value	Hexadecimal value
MQCTLO_VERSION_1	1	X'00000001'
MQCTLO_CURRENT_VERSION	1	X'00000001'

MQCTL options Consumer Control Options

Table 128. Values of constants		
Name	Decimal value	Hexadecimal value
MQCTLO_NONE	0	X'00000000'
MQCTLO_THREAD_AFFINITY	1	X'00000001'
MQCTLO_FAIL_IF QUIESCING	8192	X'00002000'

MQCUOWC_* (CICS information header Unit-of-Work Controls)



Table 129. Values of constants		
Name	Decimal value	Hexadecimal value
MQCUOWC_ONLY	273	X'00000111'
MQCUOWC_CONTINUE	65536	X'00010000'
MQCUOWC_FIRST	17	X'00000011'
MQCUOWC_MIDDLE	16	X'00000010'
MQCUOWC_LAST	272	X'00000110'
MQCUOWC_COMMIT	256	X'00000100'
MQCUOWC_BACKOUT	4352	X'00001100'

MQCXP_* (Channel exit parameter structure)

Table 130. Structures of constants	
Name	Structure
MQCXP_STRUC_ID	"CXP¬"
MQCXP_STRUC_ID_ARRAY	'C', 'X', 'P', '¬'

Note: The symbol ¬ represents a single blank character.

Table 131. Values of constants		
Name	Decimal value	Hexadecimal value
MQCXP_VERSION_1	1	X'00000001'
MQCXP_VERSION_2	2	X'00000002'
MQCXP_VERSION_3	3	X'00000003'
MQCXP_VERSION_4	4	X'00000004'
MQCXP_VERSION_5	5	X'00000005'
MQCXP_VERSION_6	6	X'00000006'
MQCXP_VERSION_7	7	X'00000007'
MQCXP_VERSION_8	8	X'00000008'
MQCXP_VERSION_9	9	X'00000009'

Table 131. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
 MQCXP_CURRENT_VERSION	8	X'00000008'
 MQCXP_CURRENT_VERSION	9	X'00000009'

MQDC_* (Destination Class)

Table 132. Values of constants		
Name	Decimal value	Hexadecimal value
MQDC_MANAGED	1	X'00000001'
MQDC_PROVIDED	2	X'00000002'

MQDCC_* (Conversion Options, and Masks and Factors)

Conversion Options

Table 133. Values of constants		
Name	Decimal value	Hexadecimal value
MQDCC_DEFAULT_CONVERSION	1	X'00000001'
MQDCC_FILL_TARGET_BUFFER	2	X'00000002'
MQDCC_INT_DEFAULT_CONVERSION	4	X'00000004'
MQDCC_SOURCE_ENC_NATIVE	(value differs by platform or version)	
MQDCC_SOURCE_ENC_NORMAL	16	X'00000010'
MQDCC_SOURCE_ENC_REVERSED	32	X'00000020'
MQDCC_SOURCE_ENC_UNDEFINED	0	X'00000000'
MQDCC_TARGET_ENC_NATIVE	(value differs by platform or version)	
MQDCC_TARGET_ENC_NORMAL	256	X'00000100'
MQDCC_TARGET_ENC_REVERSED	512	X'00000200'
MQDCC_TARGET_ENC_UNDEFINED	0	X'00000000'
MQDCC_NONE	0	X'00000000'

Conversion Options Masks and Factors

Table 134. Values of constants		
Name	Decimal value	Hexadecimal value
MQDCC_SOURCE_ENC_MASK	240	X'000000F0'
MQDCC_TARGET_ENC_MASK	3840	X'00000F00'
MQDCC_SOURCE_ENC_FACTOR	16	X'00000010'
MQDCC_TARGET_ENC_FACTOR	256	X'00000100'

MQDELO_* (Publish/Subscribe Delete Options)

Table 135. Values of constants		
Name	Decimal value	Hexadecimal value
MQDELO_NONE	0	X'00000000'
MQDELO_LOCAL	4	X'00000004'

MQDH_* (Distribution header structure)

Table 136. Structures of constants	
Name	Structure
MQDH_STRUC_ID	"DH¬¬"
MQDH_STRUC_ID_ARRAY	'D','H','¬','¬'

Note: The symbol ¬ represents a single blank character.

Table 137. Values of constants		
Name	Decimal value	Hexadecimal value
MQDH_VERSION_1	1	X'00000001'
MQDH_CURRENT_VERSION	1	X'00000001'

MQDHF_* (Distribution header Flags)

Table 138. Values of constants		
Name	Decimal value	Hexadecimal value
MQDHF_NEW_MSG_IDS	1	X'00000001'
MQDHF_NONE	0	X'00000000'

MQDISCONNECT_* (Command format Disconnect Types)

Table 139. Values of constants		
Name	Decimal value	Hexadecimal value
MQDISCONNECT_NORMAL	0	X'00000000'
MQDISCONNECT_IMPLICIT	1	X'00000001'
MQDISCONNECT_Q_MGR	2	X'00000002'

MQDL_* (Distribution Lists)

Table 140. Values of constants		
Name	Decimal value	Hexadecimal value
MQDL_SUPPORTED	1	X'00000001'
MQDL_NOT_SUPPORTED	0	X'00000000'

MQDLH_* (Dead-letter header structure)

Table 141. Structures of constants	
Name	Structure
MQDLH_STRUC_ID	"DLH¬"
MQDLH_STRUC_ID_ARRAY	'D','L','H','¬'

Note: The symbol ↵ represents a single blank character.

Table 142. Values of constants		
Name	Decimal value	Hexadecimal value
MQDLH_VERSION_1	1	X'00000001'
MQDLH_CURRENT_VERSION	1	X'00000001'

MQDLV_* (Persistent/Non-persistent Message Delivery)

Table 143. Values of constants		
Name	Decimal value	Hexadecimal value
MQDLV_AS_PARENT	0	X'00000000'
MQDLV_ALL	1	X'00000001'
MQDLV_ALL_DUR	2	X'00000002'
MQDLV_ALL_AVAIL	3	X'00000003'

MQDMHO_* (Delete message handle options and structure)

Delete message handle options structure

Table 144. Structures of constants	
Name	Structure
MQDMHO_STRUC_ID	"DMHO"
MQDMHO_STRUC_ID_ARRAY	'D', 'M', 'H', 'O'

Note: The symbol ↵ represents a single blank character.

Table 145. Values of constants		
Name	Decimal value	Hexadecimal value
MQDMHO_VERSION_1	1	X'00000001'
MQDMHO_CURRENT_VERSION	1	X'00000001'

Delete Message Handle Options

Table 146. Values of constants		
Name	Decimal value	Hexadecimal value
MQDMHO_NONE	0	X'00000000'

MQDMPO_* (Delete message property options and structure)

Delete message property options structure

Table 147. Structures of constants	
Name	Structure
MQDMPO_STRUC_ID	"DMPO"
MQDMPO_STRUC_ID_ARRAY	'D', 'M', 'P', 'O'

Note: The symbol ↵ represents a single blank character.

Table 148. Values of constants		
Name	Decimal value	Hexadecimal value
MQDMPO_VERSION_1	1	X'00000001'
MQDMPO_CURRENT_VERSION	1	X'00000001'

Delete Message Property Options

Table 149. Values of constants		
Name	Decimal value	Hexadecimal value
MQDMPO_DEL_FIRST	0	X'00000000'
MQDMPO_DEL_PROP_UNDER_CURSOR	1	X'00000001'
MQDMPO_NONE	0	X'00000000'

MQDNSWLM_* (DNS WLM)

Table 150. Values of constants		
Name	Decimal value	Hexadecimal value
MQDNSWLM_NO	0	X'00000000'
MQDNSWLM_YES	1	X'00000001'

MQDT_* (Destination Types)

Table 151. Values of constants		
Name	Decimal value	Hexadecimal value
MQDT_APPL	1	X'00000001'
MQDT_BROKER	2	X'00000002'

MQDXP_* (Conversion exit parameter structure)

Table 152. Structures of constants	
Name	Structure
MQDXP_STRUC_ID	"DXP¬"
MQDXP_STRUC_ID_ARRAY	'D','X','P','¬'

Note: The symbol ¬ represents a single blank character.

Table 153. Values of constants		
Name	Decimal value	Hexadecimal value
MQDXP_VERSION_1	1	X'00000001'
MQDXP_VERSION_2	2	X'00000002'
MQDXP_CURRENT_VERSION	2	X'00000002'

MQEC_* (Signal Values)

Table 154. Values of constants		
Name	Decimal value	Hexadecimal value
MQEC_MSG_ARRIVED	2	X'00000002'
MQEC_WAIT_INTERVAL_EXPIRED	3	X'00000003'

Table 154. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQEC_WAIT_CANCELED	4	X'00000004'
MQEC_Q_MGR QUIESCING	5	X'00000005'
MQEC_CONNECTION QUIESCING	6	X'00000006'

MQEI_* (Expiry)

Table 155. Values of constants		
Name	Decimal value	Hexadecimal value
MQEI_UNLIMITED	-1	X'FFFFFFFF'

MQENC_* (Encoding)

MQENC_* (Encoding)

Table 156. Values of constants by platform			
Name	Platform	Decimal value	Hexadecimal value
MQENC_NATIVE	IBM i	273	X'00000111'
	Linux	546	X'00000222'
	Linux on SPARC	273	X'00000111'
	Linux on x86	546	X'00000222'
	Solaris on SPARC	273	X'00000111'
	UNIX systems	273	X'00000111'
	Windows	546	X'00000222'
	Micro Focus COBOL on Windows	17	X'00000011'
	z/OS	785	X'00000311'

MQENC_* (Encoding Masks)

Table 157. Values of constants		
Name	Decimal value	Hexadecimal value
MQENC_INTEGER_MASK	15	X'0000000F'
MQENC_DECIMAL_MASK	240	X'000000F0'
MQENC_FLOAT_MASK	3840	X'00000F00'
MQENC_RESERVED_MASK	-4096	X'FFFFFF00'

MQENC_* (Encodings for Binary Integers)

Table 158. Values of constants		
Name	Decimal value	Hexadecimal value
MQENC_INTEGER_UNDEFINED	0	X'00000000'
MQENC_INTEGER_NORMAL	1	X'00000001'
MQENC_INTEGER_REVERSED	2	X'00000002'

MQENC_* (Encodings for Packed Decimal Integers)

Table 159. Values of constants		
Name	Decimal value	Hexadecimal value
MQENC_DECIMAL_UNDEFINED	0	X'00000000'
MQENC_DECIMAL_NORMAL	16	X'00000010'
MQENC_DECIMAL_REVERSED	32	X'00000020'

MQENC_* (Encodings for Floating Point Numbers)

Table 160. Values of constants		
Name	Decimal value	Hexadecimal value
MQENC_FLOAT_UNDEFINED	0	X'00000000'
MQENC_FLOAT_IEEE_NORMAL	256	X'00000100'
MQENC_FLOAT_IEEE_REVERSED	512	X'00000200'
MQENC_FLOAT_S390	768	X'00000300'
MQENC_FLOAT_TNS	1024	X'00000400'

MQEPH_* (Embedded command format header structure and Flags)

Embedded command format header structure

Table 161. Structures of constants	
Name	Structure
MQEPH_STRUC_ID	"EPH↵"
MQEPH_STRUC_ID_ARRAY	'E','P','H','↵'

Note: The symbol ↵ represents a single blank character.

Table 162. Values of constants		
Name	Decimal value	Hexadecimal value
MQEPH_STRUC_LENGTH_FIXED	68	X'00000044'
MQEPH_VERSION_1	1	X'00000001'
MQEPH_CURRENT_VERSION	1	X'00000001'

Embedded command format header Flags

Table 163. Values of constants		
Name	Decimal value	Hexadecimal value
MQEPH_NONE	0	X'00000000'
MQEPH_CCSID_EMBEDDED	1	X'00000001'

MQET_* (Command format Escape Types)

Table 164. Values of constants		
Name	Decimal value	Hexadecimal value
MQET_MQSC	1	X'00000001'

MQEVO_* (Command format Event Origins)

Table 165. Values of constants		
Name	Decimal value	Hexadecimal value
MQEVO_OTHER	0	X'00000000'
MQEVO_CONSOLE	1	X'00000001'
MQEVO_INIT	2	X'00000002'
MQEVO_MSG	3	X'00000003'
MQEVO_MQSET	4	X'00000004'
MQEVO_INTERNAL	5	X'00000005'

MQEVR_* (Command format Event Recording)

Table 166. Values of constants		
Name	Decimal value	Hexadecimal value
MQEVR_DISABLED	0	X'00000000'
MQEVR_ENABLED	1	X'00000001'
MQEVR_EXCEPTION	2	X'00000002'
MQEVR_NO_DISPLAY	3	X'00000003'

MQEXPI_* (Expiration Scan Interval)

Table 167. Values of constants		
Name	Decimal value	Hexadecimal value
MQEXPI_OFF	0	X'00000000'

MQFB_* (Feedback Values)

Table 168. Values of constants		
Name	Decimal value	Hexadecimal value
MQFB_NONE	0	X'00000000'
MQFB_SYSTEM_FIRST	1	X'00000001'
MQFB_QUIT	256	X'00000100'
MQFB_EXPIRATION	258	X'00000102'
MQFB_COA	259	X'00000103'
MQFB_COD	260	X'00000104'
MQFB_CHANNEL_COMPLETED	262	X'00000106'
MQFB_CHANNEL_FAIL_RETRY	263	X'00000107'
MQFB_CHANNEL_FAIL	264	X'00000108'
MQFB_APPL_CANNOT_BE_STARTED	265	X'00000109'
MQFB_TM_ERROR	266	X'0000010A'
MQFB_APPL_TYPE_ERROR	267	X'0000010B'
MQFB_STOPPED_BY_MSG_EXIT	268	X'0000010C'
MQFB_ACTIVITY	269	X'0000010D'
MQFB_XMIT_Q_MSG_ERROR	271	X'0000010F'

Table 168. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQFB_PAN	275	X'00000113'
MQFB_NAN	276	X'00000114'
MQFB_STOPPED_BY_CHAD_EXIT	277	X'00000115'
MQFB_STOPPED_BY_PUBSUB_EXIT	279	X'00000117'
MQFB_NOT_A_REPOSITORY_MSG	280	X'00000118'
MQFB_BIND_OPEN_CLUSRCVR_DEL	281	X'00000119'
MQFB_MAX_ACTIVITIES	282	X'0000011A'
MQFB_NOT_FORWARDED	283	X'0000011B'
MQFB_NOT_DELIVERED	284	X'0000011C'
MQFB_UNSUPPORTED_FORWARDING	285	X'0000011D'
MQFB_UNSUPPORTED_DELIVERY	286	X'0000011E'
MQFB_DATA_LENGTH_ZERO	291	X'00000123'
MQFB_DATA_LENGTH_NEGATIVE	292	X'00000124'
MQFB_DATA_LENGTH_TOO_BIG	293	X'00000125'
MQFB_BUFFER_OVERFLOW	294	X'00000126'
MQFB_LENGTH_OFF_BY_ONE	295	X'00000127'
MQFB_IIH_ERROR	296	X'00000128'
MQFB_NOT_AUTHORIZED_FOR_IMS	298	X'0000012A'
MQFB_IMS_ERROR	300	X'0000012C'
MQFB_IMS_FIRST	301	X'0000012D'
MQFB_IMS_LAST	399	X'0000018F'
MQFB_CICS_INTERNAL_ERROR	401	X'00000191'
MQFB_CICS_NOT_AUTHORIZED	402	X'00000192'
MQFB_CICS_BRIDGE_FAILURE	403	X'00000193'
MQFB_CICS_CORREL_ID_ERROR	404	X'00000194'
MQFB_CICS_CCSDID_ERROR	405	X'00000195'
MQFB_CICS_ENCODING_ERROR	406	X'00000196'
MQFB_CICS_CIH_ERROR	407	X'00000197'
MQFB_CICS_UOW_ERROR	408	X'00000198'
MQFB_CICS_COMMAREA_ERROR	409	X'00000199'
MQFB_CICS_APPL_NOT_STARTED	410	X'0000019A'
MQFB_CICS_APPL_ABENDED	411	X'0000019B'
MQFB_CICS_DLQ_ERROR	412	X'0000019C'
MQFB_CICS_UOW_BACKED_OUT	413	X'0000019D'
MQFB_PUBLICATIONS_ON_REQUEST	501	X'000001F5'
MQFB_SUBSCRIBER_IS_PUBLISHER	502	X'000001F6'
MQFB_MSG_SCOPE_MISMATCH	503	X'000001F7'
MQFB_SELECTOR_MISMATCH	504	X'000001F8'
MQFB_IMS_NACK_1A_REASON_FIRST	600	X'00000258'

Table 168. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQFB_IMS_NACK_1A_REASON_LAST	855	X'00000357'
MQFB_SYSTEM_LAST	65535	X'0000FFFF'
MQFB_APPL_FIRST	65536	X'00010000'
MQFB_APPL_LAST	99999999	X'3B9AC9FF'

MQFC_* (Command format Force Options)

Table 169. Values of constants		
Name	Decimal value	Hexadecimal value
MQFC_YES	1	X'00000001'
MQFC_NO	0	X'00000000'

MQFMT_* (Formats)

Table 170. Constant names and values	
Name	Value
MQFMT_NONE	"~~~~~"
MQFMT_ADMIN	"MQADMIN~"
MQFMT_CHANNEL_COMPLETED	"MQCHCOM~"
MQFMT_CICS	"MQCICS~"
MQFMT_COMMAND_1	"MQCMD1~"
MQFMT_COMMAND_2	"MQCMD2~"
MQFMT_DEAD_LETTER_HEADER	"MQDEAD~"
MQFMT_DIST_HEADER	"MQHDIST~"
MQFMT_EMBEDDED_PCF	"MQHEPCF~"
MQFMT_EVENT	"MQEVENT~"
MQFMT_IMS	"MQIMS~"
MQFMT_IMS_VAR_STRING	"MQIMSVS~"
MQFMT_MD_EXTENSION	"MQHMDE~"
MQFMT_PCF	"MQPCF~"
MQFMT_REF_MSG_HEADER	"MQHREF~"
MQFMT_RF_HEADER	"MQHRF~"
MQFMT_RF_HEADER_1	"MQHRF~"
MQFMT_RF_HEADER_2	"MQHRF2~"
MQFMT_STRING	"MQSTR~"
MQFMT_TRIGGER	"MQTRIG~"
MQFMT_WORK_INFO_HEADER	"MQHWIH~"
MQFMT_XMIT_Q_HEADER	"MQXMIT~"
MQFMT_NONE_ARRAY	'~','~','~','~','~','~','~','~'
MQFMT_ADMIN_ARRAY	'M','Q','A','D','M','I','N','~'
MQFMT_CHANNEL_COMPLETED_ARRAY	'M','Q','C','H','C','O','M','~'

Table 170. Constant names and values (continued)	
Name	Value
MQFMT_CICS_ARRAY	'M','Q','C','I','C','S',' ',' '
MQFMT_COMMAND_1_ARRAY	'M','Q','C','M','D','1',' ',' '
MQFMT_COMMAND_2_ARRAY	'M','Q','C','M','D','2',' ',' '
MQFMT_DEAD_LETTER_HEADER_ARRAY	'M','Q','D','E','A','D',' ',' '
MQFMT_DIST_HEADER_ARRAY	'M','Q','H','D','I','S','T',' '
MQFMT_EMBEDDED_PCF_ARRAY	'M','Q','H','E','P','C','F',' '
MQFMT_EVENT_ARRAY	'M','Q','E','V','E','N','T',' '
MQFMT_IMS_ARRAY	'M','Q','I','M','S',' ',' '
MQFMT_IMS_VAR_STRING_ARRAY	'M','Q','I','M','S','V','S',' '
MQFMT_MD_EXTENSION_ARRAY	'M','Q','H','M','D','E',' ',' '
MQFMT_PCF_ARRAY	'M','Q','P','C','F',' ',' '
MQFMT_REF_MSG_HEADER_ARRAY	'M','Q','H','R','E','F',' ',' '
MQFMT_RF_HEADER_ARRAY	'M','Q','H','R','F',' ',' '
MQFMT_RF_HEADER_1_ARRAY	'M','Q','H','R','F',' ',' '
MQFMT_RF_HEADER_2_ARRAY	'M','Q','H','R','F','2',' ',' '
MQFMT_STRING_ARRAY	'M','Q','S','T','R',' ',' '
MQFMT_TRIGGER_ARRAY	'M','Q','T','R','I','G',' ',' '
MQFMT_WORK_INFO_HEADER_ARRAY	'M','Q','H','W','I','H',' ',' '
MQFMT_XMIT_Q_HEADER_ARRAY	'M','Q','X','M','I','T',' ',' '

Note: The symbol represents a single blank character.

MQGA_* (Group Attribute Selectors)

Table 171. Values of constants		
Name	Decimal value	Hexadecimal value
MQGA_FIRST	8001	X'00001F41'
MQGA_LAST	9000	X'00002328'

MQGACF_* (Command format Group Parameter Types)

Table 172. Values of constants		
Name	Decimal value	Hexadecimal value
MQGACF_FIRST	8001	X'00001F41'
MQGACF_COMMAND_CONTEXT	8001	X'00001F41'
MQGACF_COMMAND_DATA	8002	X'00001F42'
MQGACF_TRACE_ROUTE	8003	X'00001F43'
MQGACF_OPERATION	8004	X'00001F44'
MQGACF_ACTIVITY	8005	X'00001F45'
MQGACF_EMBEDDED_MQMD	8006	X'00001F46'
MQGACF_MESSAGE	8007	X'00001F47'
MQGACF_MQMD	8008	X'00001F48'

Table 172. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQGACF_VALUE_NAMING	8009	X'00001F49'
MQGACF_Q_ACCOUNTING_DATA	8010	X'00001F4A'
MQGACF_Q_STATISTICS_DATA	8011	X'00001F4B'
MQGACF_CHL_STATISTICS_DATA	8012	X'00001F4C'
MQGACF_LAST_USED	8012	X'00001F4C'

MQGI_* (Group Identifier)

Table 173. Constant names and values	
Name	Value
MQGI_NONE	X'00...00' (24 nulls)
MQGI_NONE_ARRAY	'\0', '\0', ... (24 nulls)

MQGMO_* (Get message options and structure)

Get message options structure

Table 174. Structures of constants	
Name	Structure
MQGMO_STRUC_ID	"GMO↵"
MQGMO_STRUC_ID_ARRAY	'G', 'M', 'O', '↵'

Note: The symbol ↵ represents a single blank character.

Table 175. Values of constants		
Name	Decimal value	Hexadecimal value
MQGMO_VERSION_1	1	X'00000001'
MQGMO_VERSION_2	2	X'00000002'
MQGMO_VERSION_3	3	X'00000003'
MQGMO_VERSION_4	4	X'00000004'
MQGMO_CURRENT_VERSION	4	X'00000004'

Get Message Options

Table 176. Values of constants		
Name	Decimal value	Hexadecimal value
MQGMO_WAIT	1	X'00000001'
MQGMO_NO_WAIT	0	X'00000000'
MQGMO_SET_SIGNAL	8	X'00000008'
MQGMO_FAIL_IF QUIESCING	8192	X'00002000'
MQGMO_SYNCPOINT	2	X'00000002'
MQGMO_SYNCPOINT_IF_PERSISTENT	4096	X'00001000'
MQGMO_NO_SYNCPOINT	4	X'00000004'
MQGMO_MARK_SKIP_BACKOUT	128	X'00000080'

Table 176. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQGMO_BROWSE_FIRST	16	X'00000010'
MQGMO_BROWSE_NEXT	32	X'00000020'
MQGMO_BROWSE_MSG_UNDER_CURSOR	2048	X'00000800'
MQGMO_BROWSE_HANDLE	17825808	X'01100010'
MQGMO_BROWSE_CO_OP	18874384	X'01200010'
MQGMO_MSG_UNDER_CURSOR	256	X'00000100'
MQGMO_LOCK	512	X'00000200'
MQGMO_UNLOCK	1024	X'00000400'
MQGMO_ACCEPT_TRUNCATED_MSG	64	X'00000040'
MQGMO_CONVERT	16384	X'00004000'
MQGMO_LOGICAL_ORDER	32768	X'00008000'
MQGMO_COMPLETE_MSG	65536	X'00010000'
MQGMO_ALL_MSGS_AVAILABLE	131072	X'00020000'
MQGMO_ALL_SEGMENTS_AVAILABLE	262144	X'00040000'
MQGMO_MARK_BROWSE_HANDLE	1048576	X'00100000'
MQGMO_MARK_BROWSE_CO_OP	2097152	X'00200000'
MQGMO_UNMARK_BROWSE_CO_OP	4194304	X'00400000'
MQGMO_UNMARK_BROWSE_HANDLE	8388608	X'00800000'
MQGMO_UNMARKED_BROWSE_MSG	16777216	X'01000000'
MQGMO_PROPERTIES_FORCE_MQRFH2	33554432	X'02000000'
MQGMO_NO_PROPERTIES	67108864	X'04000000'
MQGMO_PROPERTIES_IN_HANDLE	134217728	X'08000000'
MQGMO_PROPERTIES_COMPATIBILITY	268435456	X'10000000'
MQGMO_PROPERTIES_AS_Q_DEF	0	X'00000000'
MQGMO_NONE	0	X'00000000'

MQGS_* (Group Status)

Table 177. Constant names and values	
Name	Value
MQGS_NOT_IN_GROUP	'␣'
MQGS_MSG_IN_GROUP	'G'
MQGS_LAST_MSG_IN_GROUP	'L'

Note: The symbol ␣ represents a single blank character.

MQHA_* (Handle Selectors)

Table 178. Values of constants		
Name	Decimal value	Hexadecimal value
MQHA_FIRST	4001	X'0000FA1'
MQHA_BAG_HANDLE	4001	X'0000FA1'

Table 178. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQHA_LAST_USED	4001	X'00000FA1'
MQHA_LAST	6000	X'00001770'

MQHB_* (Bag Handles)

Table 179. Values of constants		
Name	Decimal value	Hexadecimal value
MQHB_UNUSABLE_HBAG	-1	X'FFFFFFFF'
MQHB_NONE	-2	X'FFFFFFFE'

MQHC_* (Connection Handles)

Table 180. Values of constants		
Name	Decimal value	Hexadecimal value
MQHC_DEF_HCONN	0	X'00000000'
MQHC_UNUSABLE_HCONN	-1	X'FFFFFFFF'
MQHC_UNASSOCIATED_HCONN	-3	X'FFFFFFFD'

MQHM_* (Message handle)

Table 181. Values of constants		
Name	Decimal value	Hexadecimal value
MQHM_UNUSABLE_HMSG	-1	X'FFFFFFFF'
MQHM_NONE	0	X'00000000'

MQHO_* (Object Handle)

Table 182. Values of constants		
Name	Decimal value	Hexadecimal value
MQHO_UNUSABLE_HOBJ	-1	X'FFFFFFFF'
MQHO_NONE	0	X'00000000'

MQHSTATE_* (Command format Handle States)

Table 183. Values of constants		
Name	Decimal value	Hexadecimal value
MQHSTATE_INACTIVE	0	X'00000000'
MQHSTATE_ACTIVE	1	X'00000001'

MQIA_* (Integer Attribute Selectors)

Table 184. Values of constants		
Name	Decimal value	Hexadecimal value
MQIA_ACCOUNTING_CONN_OVERRIDE	136	X'00000088'
MQIA_ACCOUNTING_INTERVAL	135	X'00000087'
MQIA_ACCOUNTING_MQI	133	X'00000085'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_ACCOUNTING_Q	134	X'00000086'
MQIA_ACTIVE_CHANNELS	100	X'00000064'
MQIA_ACTIVITY_CONN_OVERRIDE	239	X'000000EF'
MQIA_ACTIVITY_RECORDING	138	X'0000008A'
MQIA_ACTIVITY_TRACE	240	X'000000F0'
MQIA_ADOPTNEWMCA_CHECK	102	X'00000066'
MQIA_ADOPTNEWMCA_TYPE	103	X'00000067'
MQIA_ADOPTNEWMCA_INTERVAL	104	X'00000068'
MQIA_APPL_TYPE	1	X'00000001'
MQIA_ADOPT_CONTEXT	260	X'00000104'
MQIA_AMQP_CAPABILITY	265	X'00000109'
MQIA_ARCHIVE	60	X'0000003C'
MQIA_AUTHENTICATION_FAIL_DELAY	259	X'00000103'
MQIA_AUTHENTICATION_METHOD	266	X'0000010A'
MQIA_AUTH_INFO_TYPE	66	X'00000042'
MQIA_AUTHORITY_EVENT	47	X'0000002F'
MQIA_AUTO_REORG_INTERVAL	174	X'000000AE'
MQIA_AUTO_REORGANIZATION	173	X'000000AD'
MQIA_BACKOUT_THRESHOLD	22	X'00000016'
MQIA_BASE_TYPE	193	X'000000C1'
MQIA_BATCH_INTERFACE_AUTO	86	X'00000056'
MQIA_BRIDGE_EVENT	74	X'0000004A'
MQIA_CF_LEVEL	70	X'00000046'
MQIA_CF_RECOVER	71	X'00000047'
MQIA_CHANNEL_AUTO_DEF	55	X'00000037'
MQIA_CHANNEL_AUTO_DEF_EVENT	56	X'00000038'
MQIA_CHANNEL_EVENT	73	X'00000049'
MQIA_CHECK_CLIENT_BINDING	258	X'00000102'
MQIA_CHECK_LOCAL_BINDING	257	X'00000101'
MQIA_CHINIT_ADAPTERS	101	X'00000065'
MQIA_CHINIT_CONTROL	119	X'00000077'
MQIA_CHINIT_DISPATCHERS	105	X'00000069'
MQIA_CHINIT_TRACE_AUTO_START	117	X'00000075'
MQIA_CHINIT_TRACE_TABLE_SIZE	118	X'00000076'
MQIA_CLUSTER_OBJECT_STATE	256	X'00000100'
MQIA_CLUSTER_PUB_ROUTE	255	X'000000FF'
MQIA_CLUSTER_Q_TYPE	59	X'0000003B'
MQIA_CLUSTER_WORKLOAD_LENGTH	58	X'0000003A'
MQIA_CLWL_MRU_CHANNELS	97	X'00000061'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_CLWL_Q_RANK	95	X'0000005F'
MQIA_CLWL_Q_PRIORITY	96	X'00000060'
MQIA_CLWL_USEQ	98	X'00000062'
MQIA_CMD_SERVER_AUTO	87	X'00000057'
MQIA_CMD_SERVER_CONTROL	120	X'00000078'
MQIA_CMD_SERVER_CONVERT_MSG	88	X'00000058'
MQIA_CMD_SERVER_DLQ_MSG	89	X'00000059'
MQIA_CODED_CHAR_SET_ID	2	X'00000002'
MQIA_COMM_EVENT	232	X'000000E8'
MQIA_COMMAND_EVENT	99	X'00000063'
MQIA_COMMAND_LEVEL	31	X'0000001F'
MQIA_CONFIGURATION_EVENT	51	X'00000033'
MQIA_CPI_LEVEL	27	X'0000001B'
MQIA_CURRENT_Q_DEPTH	3	X'00000003'
MQIA_DEF_BIND	61	X'0000003D'
MQIA_DEF_CLUSTER_XMIT_Q_TYPE	250	X'000000FA'
MQIA_DEF_INPUT_OPEN_OPTION	4	X'00000004'
MQIA_DEF_PERSISTENCE	5	X'00000005'
MQIA_DEF_PRIORITY	6	X'00000006'
MQIA_DEF_PUT_RESPONSE_TYPE	184	X'000000B8'
MQIA_DEF_READ_AHEAD	188	X'000000BC'
MQIA_DEFINITION_TYPE	7	X'00000007'
MQIA_DISPLAY_TYPE	262	X'00000106'
MQIA_DIST_LISTS	34	X'00000022'
MQIA_DNS_WLM	106	X'0000006A'
MQIA_DURABLE_SUB	175	X'000000AF'
MQIA_EXPIRY_INTERVAL	39	X'00000027'
MQIA_FIRST	1	X'00000001'
MQIA_GROUP_UR	221	X'000000DD'
MQIA_HARDEN_GET_BACKOUT	8	X'00000008'
MQIA_HIGH_Q_DEPTH	36	X'00000024'
MQIA_IGQ_PUT_AUTHORITY	65	X'00000041'
MQIA_INDEX_TYPE	57	X'00000039'
MQIA_INHIBIT_EVENT	48	X'00000030'
MQIA_INHIBIT_GET	9	X'00000009'
MQIA_INHIBIT_PUB	181	X'000000B5'
MQIA_INHIBIT_PUT	10	X'0000000A'
MQIA_INHIBIT_SUB	182	X'000000B6'
MQIA_INTRA_GROUP_queuing	64	X'00000040'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_IP_ADDRESS_VERSION	93	X'0000005D'
MQIA_LAST	2000	X'000007D0'
MQIA_LAST_USED	266	X'0000010A'
MQIA_LDAP_AUTHORMD	263	X'00000107'
MQIA_LDAP_NESTGRP	264	X'00000108'
MQIA_LDAP_SECURE_COMM	261	X'00000105'
MQIA_LISTENER_PORT_NUMBER	85	X'00000055'
MQIA_LISTENER_TIMER	107	X'0000006B'
MQIA_LOGGER_EVENT	94	X'0000005E'
MQIA_LU62_CHANNELS	108	X'0000006C'
MQIA_LOCAL_EVENT	49	X'00000031'
MQIA_MSG_MARK_BROWSE_INTERVAL	68	X'00000044'
MQIA_MAX_CHANNELS	109	X'0000006D'
MQIA_MAX_CLIENTS	172	X'000000AC'
MQIA_MAX_GLOBAL_LOCKS	83	X'00000053'
MQIA_MAX_HANDLES	11	X'0000000B'
MQIA_MAX_LOCAL_LOCKS	84	X'00000054'
MQIA_MAX_MSG_LENGTH	13	X'0000000D'
MQIA_MAX_OPEN_Q	80	X'00000050'
MQIA_MAX_PRIORITY	14	X'0000000E'
MQIA_MAX_PROPERTIES_LENGTH	192	X'000000C0'
MQIA_MAX_Q_DEPTH	15	X'0000000F'
MQIA_MAX_Q_TRIGGERS	90	X'0000005A'
MQIA_MAX_RECOVERY_TASKS	171	X'000000AB'
MQIA_MAX_UNCOMMITTED_MSGS	33	X'00000021'
MQIA_MCAST_BRIDGE	233	X'000000E9'
MQIA_MONITOR_INTERVAL	81	X'00000051'
MQIA_MONITORING_AUTO_CLUSSDR	124	X'0000007C'
MQIA_MONITORING_CHANNEL	122	X'0000007A'
MQIA_MONITORING_Q	123	X'0000007B'
MQIA_MSG_DELIVERY_SEQUENCE	16	X'00000010'
MQIA_MSG_DEQ_COUNT	38	X'00000026'
MQIA_MSG_ENQ_COUNT	37	X'00000025'
MQIA_NAME_COUNT	19	X'00000013'
MQIA_NAMELIST_TYPE	72	X'00000048'
MQIA_NPM_CLASS	78	X'0000004E'
MQIA_NPM_DELIVERY	196	X'000000C4'
MQIA_OPEN_INPUT_COUNT	17	X'00000011'
MQIA_OPEN_OUTPUT_COUNT	18	X'00000012'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_OUTBOUND_PORT_MAX	140	X'0000008C'
MQIA_OUTBOUND_PORT_MIN	110	X'0000006E'
MQIA_PAGESET_ID	62	X'0000003E'
MQIA_PERFORMANCE_EVENT	53	X'00000035'
MQIA_PLATFORM	32	X'00000020'
MQIA_PM_DELIVERY	195	X'000000C3'
MQIA_PROPERTY_CONTROL	190	X'000000BE'
MQIA_PROT_POLICY_CAPABILITY	251	X'000000FB'
MQIA_PROXY_SUB	199	X'000000C7'
MQIA_PUB_COUNT	215	X'000000D7'
MQIA_PUB_SCOPE	219	X'000000DB'
MQIA_PUBSUB_CLUSTER	249	X'000000F9'
MQIA_PUBSUB_MAXMSG_RETRY_COUNT	206	X'000000CE'
MQIA_PUBSUB_MODE	187	X'000000BB'
MQIA_PUBSUB_NP_MSG	203	X'000000CB'
MQIA_PUBSUB_NP_RESP	205	X'000000CD'
MQIA_PUBSUB_SYNC_PT	207	X'000000CF'
MQIA_Q_DEPTH_HIGH_EVENT	43	X'0000002B'
MQIA_Q_DEPTH_HIGH_LIMIT	40	X'00000028'
MQIA_Q_DEPTH_LOW_EVENT	44	X'0000002C'
MQIA_Q_DEPTH_LOW_LIMIT	41	X'00000029'
MQIA_Q_DEPTH_MAX_EVENT	42	X'0000002A'
MQIA_Q_SERVICE_INTERVAL	54	X'00000036'
MQIA_Q_SERVICE_INTERVAL_EVENT	46	X'0000002E'
MQIA_Q_TYPE	20	X'00000014'
MQIA_Q_USERS	82	X'00000052'
MQIA_QMGR_CFCONLOS	245	X'000000F5'
MQIA_QMOPT_CONS_COMMS_MSGS	155	X'0000009B'
MQIA_QMOPT_CONS_CRITICAL_MSGS	154	X'0000009A'
MQIA_QMOPT_CONS_ERROR_MSGS	153	X'00000099'
MQIA_QMOPT_CONS_INFO_MSGS	151	X'00000097'
MQIA_QMOPT_CONS_REORG_MSGS	156	X'0000009C'
MQIA_QMOPT_CONS_SYSTEM_MSGS	157	X'0000009D'
MQIA_QMOPT_CONS_WARNING_MSGS	152	X'00000098'
MQIA_QMOPT_CSMT_ON_ERROR	150	X'00000096'
MQIA_QMOPT_INTERNAL_DUMP	170	X'000000AA'
MQIA_QMOPT_LOG_COMMS_MSGS	162	X'000000A2'
MQIA_QMOPT_LOG_CRITICAL_MSGS	161	X'000000A1'
MQIA_QMOPT_LOG_ERROR_MSGS	160	X'000000A0'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_QMOPT_LOG_INFO_MSGS	158	X'0000009E'
MQIA_QMOPT_LOG_REORG_MSGS	163	X'000000A3'
MQIA_QMOPT_LOG_SYSTEM_MSGS	164	X'000000A4'
MQIA_QMOPT_LOG_WARNING_MSGS	159	X'0000009F'
MQIA_QMOPT_TRACE_COMMS	166	X'000000A6'
MQIA_QMOPT_TRACE_CONVERSION	168	X'000000A8'
MQIA_QMOPT_TRACE_REORG	167	X'000000A7'
MQIA_QMOPT_TRACE_MQI_CALLS	165	X'000000A5'
MQIA_QMOPT_TRACE_SYSTEM	169	X'000000A9'
MQIA_QSG_DISP	63	X'0000003F'
MQIA_READ_AHEAD	189	X'000000BD'
MQIA_RECEIVE_TIMEOUT	111	X'0000006F'
MQIA_RECEIVE_TIMEOUT_MIN	113	X'00000071'
MQIA_RECEIVE_TIMEOUT_TYPE	112	X'00000070'
MQIA_REMOTE_EVENT	50	X'00000032'
MQIA_RETENTION_INTERVAL	21	X'00000015'
MQIA_REVERSE_DNS_LOOKUP	254	X'000000FE'
MQIA_SCOPE	45	X'0000002D'
MQIA_SECURITY_CASE	141	X'0000008D'
MQIA_SERVICE_CONTROL	139	X'0000008B'
MQIA_SERVICE_TYPE	121	X'00000079'
MQIA_SHAREABILITY	23	X'00000017'
MQIA_SHARED_Q_Q_MGR_NAME	77	X'0000004D'
MQIA_SSL_EVENT	75	X'0000004B'
MQIA_SSL_FIPS_REQUIRED	92	X'0000005C'
MQIA_SSL_RESET_COUNT	76	X'0000004C'
MQIA_SSL_TASKS	69	X'00000045'
MQIA_START_STOP_EVENT	52	X'00000034'
MQIA_STATISTICS_CHANNEL	129	X'00000081'
MQIA_STATISTICS_AUTO_CLUSSDR	130	X'00000082'
MQIA_STATISTICS_INTERVAL	131	X'00000083'
MQIA_STATISTICS_MQI	127	X'0000007F'
MQIA_STATISTICS_Q	128	X'00000080'
MQIA_SUB_COUNT	204	X'000000CC'
MQIA_SUB_SCOPE	218	X'000000DA'
MQIA_SYNCPOINT	30	X'0000001E'
MQIA_TCP_CHANNELS	114	X'00000072'
MQIA_TCP_KEEP_ALIVE	115	X'00000073'
MQIA_TCP_STACK_TYPE	116	X'00000074'

Table 184. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIA_TIME_SINCE_RESET	35	X'00000023'
MQIA_TOPIC_DEF_PERSISTENCE	185	X'000000B9'
MQIA_TOPIC_NODE_COUNT	253	X'000000FD'
MQIA_TOPIC_TYPE	208	X'000000D0'
MQIA_TRACE_ROUTE_RECORDING	137	X'00000089'
MQIA_TREE_LIFE_TIME	183	X'000000B7'
MQIA_TRIGGER_CONTROL	24	X'00000018'
MQIA_TRIGGER_DEPTH	29	X'0000001D'
MQIA_TRIGGER_INTERVAL	25	X'00000019'
MQIA_TRIGGER_MSG_PRIORITY	26	X'0000001A'
MQIA_TRIGGER_TYPE	28	X'0000001C'
MQIA_TRIGGER_RESTART	91	X'0000005B'
MQIA_USAGE	12	X'0000000C'
MQIA_USE_DEAD_LETTER_Q	234	X'000000EA'
MQIA_USER_LIST	2000	X'000007D0'
MQIA_WILDCARD_OPERATION	216	X'000000D8'
MQIA_XR_CAPABILITY	243	X'000000F3'

MQIACF_* (Command format Integer Parameter Types)

Table 185. Values of constants		
Name	Decimal value	Hexadecimal value
MQIACF_FIRST	1001	X'000003E9'
MQIACF_Q_MGR_ATTRS	1001	X'000003E9'
MQIACF_Q_ATTRS	1002	X'000003EA'
MQIACF_PROCESS_ATTRS	1003	X'000003EB'
MQIACF_NAMELIST_ATTRS	1004	X'000003EC'
MQIACF_FORCE	1005	X'000003ED'
MQIACF_REPLACE	1006	X'000003EE'
MQIACF_PURGE	1007	X'000003EF'
MQIACF_QUIESCE	1008	X'000003F0'
MQIACF_MODE	1008	X'000003F0'
MQIACF_ALL	1009	X'000003F1'
MQIACF_EVENT_APPL_TYPE	1010	X'000003F2'
MQIACF_EVENT_ORIGIN	1011	X'000003F3'
MQIACF_PARAMETER_ID	1012	X'000003F4'
MQIACF_ERROR_ID	1013	X'000003F5'
MQIACF_ERROR_IDENTIFIER	1013	X'000003F5'
MQIACF_SELECTOR	1014	X'000003F6'
MQIACF_CHANNEL_ATTRS	1015	X'000003F7'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_OBJECT_TYPE	1016	X'000003F8'
MQIACF_ESCAPE_TYPE	1017	X'000003F9'
MQIACF_ERROR_OFFSET	1018	X'000003FA'
MQIACF_AUTH_INFO_ATTRS	1019	X'000003FB'
MQIACF_REASON_QUALIFIER	1020	X'000003FC'
MQIACF_COMMAND	1021	X'000003FD'
MQIACF_OPEN_OPTIONS	1022	X'000003FE'
MQIACF_OPEN_TYPE	1023	X'000003FF'
MQIACF_PROCESS_ID	1024	X'00000400'
MQIACF_THREAD_ID	1025	X'00000401'
MQIACF_Q_STATUS_ATTRS	1026	X'00000402'
MQIACF_UNCOMMITTED_MSGS	1027	X'00000403'
MQIACF_HANDLE_STATE	1028	X'00000404'
MQIACF_AUX_ERROR_DATA_INT_1	1070	X'0000042E'
MQIACF_AUX_ERROR_DATA_INT_2	1071	X'0000042F'
MQIACF_CONV_REASON_CODE	1072	X'00000430'
MQIACF_BRIDGE_TYPE	1073	X'00000431'
MQIACF_INQUIRY	1074	X'00000432'
MQIACF_WAIT_INTERVAL	1075	X'00000433'
MQIACF_OPTIONS	1076	X'00000434'
MQIACF_BROKER_OPTIONS	1077	X'00000435'
MQIACF_REFRESH_TYPE	1078	X'00000436'
MQIACF_SEQUENCE_NUMBER	1079	X'00000437'
MQIACF_INTEGER_DATA	1080	X'00000438'
MQIACF_REGISTRATION_OPTIONS	1081	X'00000439'
MQIACF_PUBLICATION_OPTIONS	1082	X'0000043A'
MQIACF_CLUSTER_INFO	1083	X'0000043B'
MQIACF_Q_MGR_DEFINITION_TYPE	1084	X'0000043C'
MQIACF_Q_MGR_TYPE	1085	X'0000043D'
MQIACF_ACTION	1086	X'0000043E'
MQIACF_SUSPEND	1087	X'0000043F'
MQIACF_BROKER_COUNT	1088	X'00000440'
MQIACF_APPL_COUNT	1089	X'00000441'
MQIACF_ANONYMOUS_COUNT	1090	X'00000442'
MQIACF_REG_REG_OPTIONS	1091	X'00000443'
MQIACF_DELETE_OPTIONS	1092	X'00000444'
MQIACF_CLUSTER_Q_MGR_ATTRS	1093	X'00000445'
MQIACF_REFRESH_INTERVAL	1094	X'00000446'
MQIACF_REFRESH_REPOSITORY	1095	X'00000447'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_REMOVE_QUEUES	1096	X'00000448'
MQIACF_OPEN_INPUT_TYPE	1098	X'0000044A'
MQIACF_OPEN_OUTPUT	1099	X'0000044B'
MQIACF_OPEN_SET	1100	X'0000044C'
MQIACF_OPEN_INQUIRE	1101	X'0000044D'
MQIACF_OPEN_BROWSE	1102	X'0000044E'
MQIACF_Q_STATUS_TYPE	1103	X'0000044F'
MQIACF_Q_HANDLE	1104	X'00000450'
MQIACF_Q_STATUS	1105	X'00000451'
MQIACF_SECURITY_TYPE	1106	X'00000452'
MQIACF_CONNECTION_ATTRS	1107	X'00000453'
MQIACF_CONNECT_OPTIONS	1108	X'00000454'
MQIACF_CONN_INFO_TYPE	1110	X'00000456'
MQIACF_CONN_INFO_CONN	1111	X'00000457'
MQIACF_CONN_INFO_HANDLE	1112	X'00000458'
MQIACF_CONN_INFO_ALL	1113	X'00000459'
MQIACF_AUTH_PROFILE_ATTRS	1114	X'0000045A'
MQIACF_AUTHORIZATION_LIST	1115	X'0000045B'
MQIACF_AUTH_ADD_AUTHS	1116	X'0000045C'
MQIACF_AUTH_REMOVE_AUTHS	1117	X'0000045D'
MQIACF_ENTITY_TYPE	1118	X'0000045E'
MQIACF_COMMAND_INFO	1120	X'00000460'
MQIACF_CMDSCOPE_Q_MGR_COUNT	1121	X'00000461'
MQIACF_Q_MGR_SYSTEM	1122	X'00000462'
MQIACF_Q_MGR_EVENT	1123	X'00000463'
MQIACF_Q_MGR_DQM	1124	X'00000464'
MQIACF_Q_MGR_CLUSTER	1125	X'00000465'
MQIACF_QSG_DISPS	1126	X'00000466'
MQIACF_UOW_STATE	1128	X'00000468'
MQIACF_SECURITY_ITEM	1129	X'00000469'
MQIACF_CF_STRUC_STATUS	1130	X'0000046A'
MQIACF_UOW_TYPE	1132	X'0000046C'
MQIACF_CF_STRUC_ATTRS	1133	X'0000046D'
MQIACF_EXCLUDE_INTERVAL	1134	X'0000046E'
MQIACF_CF_STATUS_TYPE	1135	X'0000046F'
MQIACF_CF_STATUS_SUMMARY	1136	X'00000470'
MQIACF_CF_STATUS_CONNECT	1137	X'00000471'
MQIACF_CF_STATUS_BACKUP	1138	X'00000472'
MQIACF_CF_STRUC_TYPE	1139	X'00000473'


Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_CF_STRUC_SIZE_MAX	1140	X'00000474'
MQIACF_CF_STRUC_SIZE_USED	1141	X'00000475'
MQIACF_CF_STRUC_ENTRIES_MAX	1142	X'00000476'
MQIACF_CF_STRUC_ENTRIES_USED	1143	X'00000477'
MQIACF_CF_STRUC_BACKUP_SIZE	1144	X'00000478'
MQIACF_MOVE_TYPE	1145	X'00000479'
MQIACF_MOVE_TYPE_MOVE	1146	X'0000047A'
MQIACF_MOVE_TYPE_ADD	1147	X'0000047B'
MQIACF_Q_MGR_NUMBER	1148	X'0000047C'
MQIACF_Q_MGR_STATUS	1149	X'0000047D'
MQIACF_DB2_CONN_STATUS	1150	X'0000047E'
MQIACF_SECURITY_ATTRS	1151	X'0000047F'
MQIACF_SECURITY_TIMEOUT	1152	X'00000480'
MQIACF_SECURITY_INTERVAL	1153	X'00000481'
MQIACF_SECURITY_SWITCH	1154	X'00000482'
MQIACF_SECURITY_SETTING	1155	X'00000483'
MQIACF_STORAGE_CLASS_ATTRS	1156	X'00000484'
MQIACF_USAGE_TYPE	1157	X'00000485'
MQIACF_BUFFER_POOL_ID	1158	X'00000486'
MQIACF_USAGE_TOTAL_PAGES	1159	X'00000487'
MQIACF_USAGE_UNUSED_PAGES	1160	X'00000488'
MQIACF_USAGE_PERSIST_PAGES	1161	X'00000489'
MQIACF_USAGE_NONPERSIST_PAGES	1162	X'0000048A'
MQIACF_USAGE_RESTART_EXTENTS	1163	X'0000048B'
MQIACF_USAGE_EXPAND_COUNT	1164	X'0000048C'
MQIACF_PAGESET_STATUS	1165	X'0000048D'
MQIACF_USAGE_TOTAL_BUFFERS	1166	X'0000048E'
MQIACF_USAGE_DATA_SET_TYPE	1167	X'0000048F'
MQIACF_USAGE_PAGESET	1168	X'00000490'
MQIACF_USAGE_DATA_SET	1169	X'00000491'
MQIACF_USAGE_BUFFER_POOL	1170	X'00000492'
MQIACF_MOVE_COUNT	1171	X'00000493'
MQIACF_EXPIRY_Q_COUNT	1172	X'00000494'
MQIACF_CONFIGURATION_OBJECTS	1173	X'00000495'
MQIACF_CONFIGURATION_EVENTS	1174	X'00000496'
MQIACF_SYSP_TYPE	1175	X'00000497'
MQIACF_SYSP_DEALLOC_INTERVAL	1176	X'00000498'
MQIACF_SYSP_MAX_ARCHIVE	1177	X'00000499'
MQIACF_SYSP_MAX_READ_TAPES	1178	X'0000049A'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_SYSP_IN_BUFFER_SIZE	1179	X'0000049B'
MQIACF_SYSP_OUT_BUFFER_SIZE	1180	X'0000049C'
MQIACF_SYSP_OUT_BUFFER_COUNT	1181	X'0000049D'
MQIACF_SYSP_ARCHIVE	1182	X'0000049E'
MQIACF_SYSP_DUAL_ACTIVE	1183	X'0000049F'
MQIACF_SYSP_DUAL_ARCHIVE	1184	X'000004A0'
MQIACF_SYSP_DUAL_BSDS	1185	X'000004A1'
MQIACF_SYSP_MAX_CONNS	1186	X'000004A2'
MQIACF_SYSP_MAX_CONNS_FORE	1187	X'000004A3'
MQIACF_SYSP_MAX_CONNS_BACK	1188	X'000004A4'
MQIACF_SYSP_EXIT_INTERVAL	1189	X'000004A5'
MQIACF_SYSP_EXIT_TASKS	1190	X'000004A6'
MQIACF_SYSP_CHKPOINT_COUNT	1191	X'000004A7'
MQIACF_SYSP_OTMA_INTERVAL	1192	X'000004A8'
MQIACF_SYSP_Q_INDEX_DEFER	1193	X'000004A9'
MQIACF_SYSP_DB2_TASKS	1194	X'000004AA'
MQIACF_SYSP_RESLEVEL_AUDIT	1195	X'000004AB'
MQIACF_SYSP_ROUTING_CODE	1196	X'000004AC'
MQIACF_SYSP_SMF_ACCOUNTING	1197	X'000004AD'
MQIACF_SYSP_SMF_STATS	1198	X'000004AE'
MQIACF_SYSP_SMF_INTERVAL	1199	X'000004AF'
MQIACF_SYSP_TRACE_CLASS	1200	X'000004B0'
MQIACF_SYSP_TRACE_SIZE	1201	X'000004B1'
MQIACF_SYSP_WLM_INTERVAL	1202	X'000004B2'
MQIACF_SYSP_ALLOC_UNIT	1203	X'000004B3'
MQIACF_SYSP_ARCHIVE_RETAIN	1204	X'000004B4'
MQIACF_SYSP_ARCHIVE_WTOR	1205	X'000004B5'
MQIACF_SYSP_BLOCK_SIZE	1206	X'000004B6'
MQIACF_SYSP_CATALOG	1207	X'000004B7'
MQIACF_SYSP_COMPACT	1208	X'000004B8'
MQIACF_SYSP_ALLOC_PRIMARY	1209	X'000004B9'
MQIACF_SYSP_ALLOC_SECONDARY	1210	X'000004BA'
MQIACF_SYSP_PROTECT	1211	X'000004BB'
MQIACF_SYSP_QUIESCE_INTERVAL	1212	X'000004BC'
MQIACF_SYSP_TIMESTAMP	1213	X'000004BD'
MQIACF_SYSP_UNIT_ADDRESS	1214	X'000004BE'
MQIACF_SYSP_UNIT_STATUS	1215	X'000004BF'
MQIACF_SYSP_LOG_COPY	1216	X'000004C0'
MQIACF_SYSP_LOG_USED	1217	X'000004C1'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_SYSP_LOG_SUSPEND	1218	X'000004C2'
MQIACF_SYSP_OFFLOAD_STATUS	1219	X'000004C3'
MQIACF_SYSP_TOTAL_LOGS	1220	X'000004C4'
MQIACF_SYSP_FULL_LOGS	1221	X'000004C5'
MQIACF_LISTENER_ATTRS	1222	X'000004C6'
MQIACF_LISTENER_STATUS_ATTRS	1223	X'000004C7'
MQIACF_SERVICE_ATTRS	1224	X'000004C8'
MQIACF_SERVICE_STATUS_ATTRS	1225	X'000004C9'
MQIACF_Q_TIME_INDICATOR	1226	X'000004CA'
MQIACF_OLDEST_MSG_AGE	1227	X'000004CB'
MQIACF_AUTH_OPTIONS	1228	X'000004CC'
MQIACF_Q_MGR_STATUS_ATTRS	1229	X'000004CD'
MQIACF_CONNECTION_COUNT	1230	X'000004CE'
MQIACF_Q_MGR_FACILITY	1231	X'000004CF'
MQIACF_CHINIT_STATUS	1232	X'000004D0'
MQIACF_CMD_SERVER_STATUS	1233	X'000004D1'
MQIACF_ROUTE_DETAIL	1234	X'000004D2'
MQIACF_RECORDED_ACTIVITIES	1235	X'000004D3'
MQIACF_MAX_ACTIVITIES	1236	X'000004D4'
MQIACF_DISCONTINUITY_COUNT	1237	X'000004D5'
MQIACF_ROUTE_ACCUMULATION	1238	X'000004D6'
MQIACF_ROUTE_DELIVERY	1239	X'000004D7'
MQIACF_OPERATION_TYPE	1240	X'000004D8'
MQIACF_BACKOUT_COUNT	1241	X'000004D9'
MQIACF_COMP_CODE	1242	X'000004DA'
MQIACF_ENCODING	1243	X'000004DB'
MQIACF_EXPIRY	1244	X'000004DC'
MQIACF_FEEDBACK	1245	X'000004DD'
MQIACF_MSG_FLAGS	1247	X'000004DF'
MQIACF_MSG_LENGTH	1248	X'000004E0'
MQIACF_MSG_TYPE	1249	X'000004E1'
MQIACF_OFFSET	1250	X'000004E2'
MQIACF_ORIGINAL_LENGTH	1251	X'000004E3'
MQIACF_PERSISTENCE	1252	X'000004E4'
MQIACF_PRIORITY	1253	X'000004E5'
MQIACF_REASON_CODE	1254	X'000004E6'
MQIACF_REPORT	1255	X'000004E7'
MQIACF_VERSION	1256	X'000004E8'
MQIACF_UNRECORDED_ACTIVITIES	1257	X'000004E9'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_MONITORING	1258	X'000004EA'
MQIACF_ROUTE_FORWARDING	1259	X'000004EB'
MQIACF_SERVICE_STATUS	1260	X'000004EC'
MQIACF_Q_TYPES	1261	X'000004ED'
MQIACF_USER_ID_SUPPORT	1262	X'000004EE'
MQIACF_INTERFACE_VERSION	1263	X'000004EF'
MQIACF_AUTH_SERVICE_ATTRS	1264	X'000004F0'
MQIACF_USAGE_EXPAND_TYPE	1265	X'000004F1'
MQIACF_SYSP_CLUSTER_CACHE	1266	X'000004F2'
MQIACF_SYSP_DB2_BLOB_TASKS	1267	X'000004F3'
MQIACF_SYSP_WLM_INT_UNITS	1268	X'000004F4'
MQIACF_TOPIC_ATTRS	1269	X'000004F5'
MQIACF_PUBSUB_PROPERTIES	1271	X'000004F7'
MQIACF_DESTINATION_CLASS	1273	X'000004F9'
MQIACF_DURABLE_SUBSCRIPTION	1274	X'000004FA'
MQIACF_SUBSCRIPTION_SCOPE	1275	X'000004FB'
MQIACF_VARIABLE_USER_ID	1277	X'000004FD'
MQIACF_REQUEST_ONLY	1280	X'00000500'
MQIACF_PUB_PRIORITY	1283	X'00000503'
MQIACF_SUB_ATTRS	1287	X'00000507'
MQIACF_WILDCARD_SCHEMA	1288	X'00000508'
MQIACF_SUB_TYPE	1289	X'00000509'
MQIACF_MESSAGE_COUNT	1290	X'0000050A'
MQIACF_Q_MGR_PUBSUB	1291	X'0000050B'
MQIACF_Q_MGR_VERSION	1292	X'0000050C'
MQIACF_SUB_STATUS_ATTRS	1294	X'0000050E'
MQIACF_TOPIC_STATUS	1295	X'0000050F'
MQIACF_TOPIC_SUB	1296	X'00000510'
MQIACF_TOPIC_PUB	1297	X'00000511'
MQIACF_RETAINED_PUBLICATION	1300	X'00000514'
MQIACF_TOPIC_STATUS_ATTRS	1301	X'00000515'
MQIACF_TOPIC_STATUS_TYPE	1302	X'00000516'
MQIACF_SUB_OPTIONS	1303	X'00000517'
MQIACF_PUBLISH_COUNT	1304	X'00000518'
MQIACF_CLEAR_TYPE	1305	X'00000519'
MQIACF_CLEAR_SCOPE	1306	X'0000051A'
MQIACF_SUB_LEVEL	1307	X'0000051B'
MQIACF_ASYNC_STATE	1308	X'0000051C'
MQIACF_SUB_SUMMARY	1309	X'0000051D'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_OBSOLETE_MSGS	1310	X'0000051E'
MQIACF_PUBSUB_STATUS	1311	X'0000051F'
MQIACF_PS_STATUS_TYPE	1314	X'00000522'
MQIACF_PUBSUB_STATUS_ATTRS	1318	X'00000526'
MQIACF_SELECTOR_TYPE	1321	X'00000529'
MQIACF_MCAST_REL_INDICATOR	1351	X'00000547'
MQIACF_CHLAUTH_TYPE	1352	X'00000548'
MQXR_DIAGNOSTICS_TYPE	1354	X'0000054A'
MQIACF_CHLAUTH_ATTRS	1355	X'0000054B'
MQIACF_OPERATION_ID	1356	X'0000054C'
MQIACF_API_CALLER_TYPE	1357	X'0000054D'
MQIACF_API_ENVIRONMENT	1358	X'0000054E'
MQIACF_TRACE_DETAIL	1359	X'0000054F'
MQIACF_HOBJ	1360	X'00000550'
MQIACF_CALL_TYPE	1361	X'00000551'
MQIACF_MQCB_OPERATION	1362	X'00000552'
MQIACF_MQCB_TYPE	1363	X'00000553'
MQIACF_MQCB_OPTIONS	1364	X'00000554'
MQIACF_CLOSE_OPTIONS	1365	X'00000555'
MQIACF_CTL_OPERATION	1366	X'00000556'
MQIACF_GET_OPTIONS	1367	X'00000557'
MQIACF_RECS_PRESENT	1368	X'00000558'
MQIACF_KNOWN_DEST_COUNT	1369	X'00000559'
MQIACF_UNKNOWN_DEST_COUNT	1370	X'0000055A'
MQIACF_INVALID_DEST_COUNT	1371	X'0000055B'
MQIACF_RESOLVED_TYPE	1372	X'0000055C'
MQIACF_PUT_OPTIONS	1373	X'0000055D'
MQIACF_BUFFER_LENGTH	1374	X'0000055E'
MQIACF_TRACE_DATA_LENGTH	1375	X'0000055F'
MQIACF_SMDS_EXPANDST	1376	X'00000560'
MQIACF_STRUC_LENGTH	1377	X'00000561'
MQIACF_ITEM_COUNT	1378	X'00000562'
MQIACF_EXPIRY_TIME	1379	X'00000563'
MQIACF_CONNECT_TIME	1380	X'00000564'
MQIACF_DISCONNECT_TIME	1381	X'00000565'
MQIACF_HSUB	1382	X'00000566'
MQIACF_SUBRQ_OPTIONS	1383	X'00000567'
MQIACF_XA_RMID	1384	X'00000568'
MQIACF_XA_FLAGS	1385	X'00000569'

Table 185. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACF_XA_RETCODE	1386	X'0000056A'
MQIACF_XA_HANDLE	1387	X'0000056B'
MQIACF_XA_RETVAL	1388	X'0000056C'
MQIACF_STATUS_TYPE	1389	X'0000056D'
MQIACF_XA_COUNT	1390	X'0000056E'
MQIACF_SELECTOR_COUNT	1391	X'0000056F'
MQIACF_SELECTORS	1392	X'00000570'
MQIACF_INTATTR_COUNT	1393	X'00000571'
MQIACF_INTATTRS	1394	X'00000572'
MQIACF_SUBRQ_ACTION	1395	X'00000573'
MQIACF_NUM_PUBS	1396	X'00000574'
MQIACF_POINTER_SIZE	1397	X'00000575'
MQIACF_REMOVE_AUTHREC	1398	X'00000576'
MQIACF_XR_ATTRS	1399	X'00000577'
MQIACF_APPL_FUNCTION_TYPE	1400	X'00000578'
MQIACF_AMQP_ATTRS	1401	X'00000579'
MQIACF_EXPORT_TYPE	1402	X'0000057A'
MQIACF_EXPORT_ATTRS	1403	X'0000057B'
MQIACF_SYSTEM_OBJECTS	1404	X'0000057C'
MQIACF_CONNECTION_SWAP	1405	X'0000057D'
MQIACF_AMQP_DIAGNOSTICS_TYPE	1406	X'0000057E'
MQIACF_BUFFER_POOL_LOCATION	1408	X'00000580'
MQIACF_LDAP_CONNECTION_STATUS	1409	X'00000581'
MQIACF_SYSP_MAX_ACE_POOL	1410	X'00000582'
MQIACF_PAGECLAS	1411	X'00000583'
 MQIACF_AUTH_REC_TYPE	1412	X'00000584'
MQIACF_LAST_USED	1412	X'00000584'

MQIACH_* (Command format Integer Channel Types)

Table 186. Values of constants		
Name	Decimal value	Hexadecimal value
MQIACH_FIRST	1501	X'000005DD'
MQIACH_XMIT_PROTOCOL_TYPE	1501	X'000005DD'
MQIACH_BATCH_SIZE	1502	X'000005DE'
MQIACH_DISC_INTERVAL	1503	X'000005DF'
MQIACH_SHORT_TIMER	1504	X'000005E0'
MQIACH_SHORT_RETRY	1505	X'000005E1'
MQIACH_LONG_TIMER	1506	X'000005E2'
MQIACH_LONG_RETRY	1507	X'000005E3'

Table 186. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQIACH_PUT_AUTHORITY	1508	X'000005E4'
MQIACH_SEQUENCE_NUMBER_WRAP	1509	X'000005E5'
MQIACH_MAX_MSG_LENGTH	1510	X'000005E6'
MQIACH_CHANNEL_TYPE	1511	X'000005E7'
MQIACH_DATA_COUNT	1512	X'000005E8'
MQIACH_NAME_COUNT	1513	X'000005E9'
MQIACH_MSG_SEQUENCE_NUMBER	1514	X'000005EA'
MQIACH_DATA_CONVERSION	1515	X'000005EB'
MQIACH_IN_DOUBT	1516	X'000005EC'
MQIACH_MCA_TYPE	1517	X'000005ED'
MQIACH_SESSION_COUNT	1518	X'000005EE'
MQIACH_ADAPTER	1519	X'000005EF'
MQIACH_COMMAND_COUNT	1520	X'000005F0'
MQIACH_SOCKET	1521	X'000005F1'
MQIACH_PORT	1522	X'000005F2'
MQIACH_CHANNEL_INSTANCE_TYPE	1523	X'000005F3'
MQIACH_CHANNEL_INSTANCE_ATTRS	1524	X'000005F4'
MQIACH_CHANNEL_ERROR_DATA	1525	X'000005F5'
MQIACH_CHANNEL_TABLE	1526	X'000005F6'
MQIACH_CHANNEL_STATUS	1527	X'000005F7'
MQIACH_INDOUBT_STATUS	1528	X'000005F8'
MQIACH_LAST_SEQ_NUMBER	1529	X'000005F9'
MQIACH_LAST_SEQUENCE_NUMBER	1529	X'000005F9'
MQIACH_CURRENT_MSGS	1531	X'000005FB'
MQIACH_CURRENT_SEQ_NUMBER	1532	X'000005FC'
MQIACH_CURRENT_SEQUENCE_NUMBER	1532	X'000005FC'
MQIACH_SSL_RETURN_CODE	1533	X'000005FD'
MQIACH_MSGS	1534	X'000005FE'
MQIACH_BYTES_SENT	1535	X'000005FF'
MQIACH_BYTES_RCVD	1536	X'00000600'
MQIACH_BYTES_RECEIVED	1536	X'00000600'
MQIACH_BATCHES	1537	X'00000601'
MQIACH_BUFFERS_SENT	1538	X'00000602'
MQIACH_BUFFERS_RCVD	1539	X'00000603'
MQIACH_BUFFERS_RECEIVED	1539	X'00000603'
MQIACH_LONG_RETRIES_LEFT	1540	X'00000604'
MQIACH_SHORT_RETRIES_LEFT	1541	X'00000605'
MQIACH_MCA_STATUS	1542	X'00000606'
MQIACH_STOP_REQUESTED	1543	X'00000607'

Table 186. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACH_MR_COUNT	1544	X'00000608'
MQIACH_MR_INTERVAL	1545	X'00000609'
MQIACH_NPM_SPEED	1562	X'0000061A'
MQIACH_HB_INTERVAL	1563	X'0000061B'
MQIACH_BATCH_INTERVAL	1564	X'0000061C'
MQIACH_NETWORK_PRIORITY	1565	X'0000061D'
MQIACH_KEEP_ALIVE_INTERVAL	1566	X'0000061E'
MQIACH_BATCH_HB	1567	X'0000061F'
MQIACH_SSL_CLIENT_AUTH	1568	X'00000620'
MQIACH_ALLOC_RETRY	1570	X'00000622'
MQIACH_ALLOC_FAST_TIMER	1571	X'00000623'
MQIACH_ALLOC_SLOW_TIMER	1572	X'00000624'
MQIACH_DISC_RETRY	1573	X'00000625'
MQIACH_PORT_NUMBER	1574	X'00000626'
MQIACH_HDR_COMPRESSION	1575	X'00000627'
MQIACH_MSG_COMPRESSION	1576	X'00000628'
MQIACH_CLWL_CHANNEL_RANK	1577	X'00000629'
MQIACH_CLWL_CHANNEL_PRIORITY	1578	X'0000062A'
MQIACH_CLWL_CHANNEL_WEIGHT	1579	X'0000062B'
MQIACH_CHANNEL_DISP	1580	X'0000062C'
MQIACH_INBOUND_DISP	1581	X'0000062D'
MQIACH_CHANNEL_TYPES	1582	X'0000062E'
MQIACH_ADAPS_STARTED	1583	X'0000062F'
MQIACH_ADAPS_MAX	1584	X'00000630'
MQIACH_DISPS_STARTED	1585	X'00000631'
MQIACH_DISPS_MAX	1586	X'00000632'
MQIACH_SSLTASKS_STARTED	1587	X'00000633'
MQIACH_SSLTASKS_MAX	1588	X'00000634'
MQIACH_CURRENT_CHL	1589	X'00000635'
MQIACH_CURRENT_CHL_MAX	1590	X'00000636'
MQIACH_CURRENT_CHL_TCP	1591	X'00000637'
MQIACH_CURRENT_CHL_LU62	1592	X'00000638'
MQIACH_ACTIVE_CHL	1593	X'00000639'
MQIACH_ACTIVE_CHL_MAX	1594	X'0000063A'
MQIACH_ACTIVE_CHL_PAUSED	1595	X'0000063B'
MQIACH_ACTIVE_CHL_STARTED	1596	X'0000063C'
MQIACH_ACTIVE_CHL_STOPPED	1597	X'0000063D'
MQIACH_ACTIVE_CHL_RETRY	1598	X'0000063E'
MQIACH_LISTENER_STATUS	1599	X'0000063F'

Table 186. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACH_SHARED_CHL_RESTART	1600	X'00000640'
MQIACH_LISTENER_CONTROL	1601	X'00000641'
MQIACH_BACKLOG	1602	X'00000642'
MQIACH_XMITQ_TIME_INDICATOR	1604	X'00000644'
MQIACH_NETWORK_TIME_INDICATOR	1605	X'00000645'
MQIACH_EXIT_TIME_INDICATOR	1606	X'00000646'
MQIACH_BATCH_SIZE_INDICATOR	1607	X'00000647'
MQIACH_XMITQ_MSGS_AVAILABLE	1608	X'00000648'
MQIACH_CHANNEL_SUBSTATE	1609	X'00000649'
MQIACH_SSL_KEY_RESETS	1610	X'0000064A'
MQIACH_COMPRESSION_RATE	1611	X'0000064B'
MQIACH_COMPRESSION_TIME	1612	X'0000064C'
MQIACH_MAX_XMIT_SIZE	1613	X'0000064D'
MQIACH_DEF_CHANNEL_DISP	1614	X'0000064E'
MQIACH_SHARING_CONVERSATIONS	1615	X'0000064F'
MQIACH_MAX_SHARING_CONVS	1616	X'00000650'
MQIACH_CURRENT_SHARING_CONVS	1617	X'00000651'
MQIACH_MAX_INSTANCES	1618	X'00000652'
MQIACH_MAX_INSTS_PER_CLIENT	1619	X'00000653'
MQIACH_CLIENT_CHANNEL_WEIGHT	1620	X'00000654'
MQIACH_CONNECTION_AFFINITY	1621	X'00000655'
MQIACH_RESET_REQUESTED	1623	X'00000657'
MQIACH_BATCH_DATA_LIMIT	1624	X'00000658'
MQIACH_MSG_HISTORY	1625	X'00000659'
MQIACH_MULTICAST_PROPERTIES	1626	X'0000065A'
MQIACH_NEW_SUBSCRIBER_HISTORY	1627	X'0000065B'
MQIACH_MC_HB_INTERVAL	1628	X'0000065C'
MQIACH_USE_CLIENT_ID	1629	X'0000065D'
MQIACH_MQTT_KEEP_ALIVE	1630	X'0000065E'
MQIACH_IN_DOUBT_IN	1631	X'0000065F'
MQIACH_IN_DOUBT_OUT	1632	X'00000660'
MQIACH_MSGS_SENT<	1633	X'00000661'
MQIACH_MSGS_RECEIVED	1634	X'00000662'
MQIACH_MSGS_RCVD	1634	X'00000662'
MQIACH_PENDING_OUT	1635	X'00000663'
MQIACH_AVAILABLE_CIPHERSPECS	1636	X'00000664'
MQIACH_MATCH	1637	X'00000665'
MQIACH_USER_SOURCE	1638	X'00000666'
MQIACH_WARNING	1639	X'00000667'

Table 186. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIACH_DEF_RECONNECT	1640	X'00000668'
MQIACH_CHANNEL_SUMMARY_ATTRS	1642	X'0000066A'
MQIACH_PROTOCOL	1643	X'0000066B'
MQIACH_AMQPKEEPALIVE	1644	X'0000066C'
MQIACH_SECURITY_PROTOCOL	1645	X'0000066D'
MQIACH_LAST_USED	1645	X'0000066D'

MQIAMO_* (Command format Integer Monitoring Parameter Types)

Table 187. Values of constants		
Name	Decimal value	Hexadecimal value
MQIAMO_FIRST	701	X'000002BD'
MQIAMO_AVG_BATCH_SIZE	702	X'000002BE'
MQIAMO_AVG_Q_TIME	703	X'000002BF'
MQIAMO_BACKOUTS	704	X'000002C0'
MQIAMO_BROWSES	705	X'000002C1'
MQIAMO_BROWSE_MAX_BYTES	706	X'000002C2'
MQIAMO_BROWSE_MIN_BYTES	707	X'000002C3'
MQIAMO_BROWSES_FAILED	708	X'000002C4'
MQIAMO_CLOSSES	709	X'000002C5'
MQIAMO_COMMITS	710	X'000002C6'
MQIAMO_COMMITS_FAILED	711	X'000002C7'
MQIAMO_CONNS	712	X'000002C8'
MQIAMO_CONNS_MAX	713	X'000002C9'
MQIAMO_DISCS	714	X'000002CA'
MQIAMO_DISCS_IMPLICIT	715	X'000002CB'
MQIAMO_DISC_TYPE	716	X'000002CC'
MQIAMO_EXIT_TIME_AVG	717	X'000002CD'
MQIAMO_EXIT_TIME_MAX	718	X'000002CE'
MQIAMO_EXIT_TIME_MIN	719	X'000002CF'
MQIAMO_FULL_BATCHES	720	X'000002D0'
MQIAMO_GENERATED_MSGS	721	X'000002D1'
MQIAMO_GETS	722	X'000002D2'
MQIAMO_GET_MAX_BYTES	723	X'000002D3'
MQIAMO_GET_MIN_BYTES	724	X'000002D4'
MQIAMO_GETS_FAILED	725	X'000002D5'
MQIAMO_INCOMPLETE_BATCHES	726	X'000002D6'
MQIAMO_INQS	727	X'000002D7'
MQIAMO_MSGS	728	X'000002D8'
MQIAMO_NET_TIME_AVG	729	X'000002D9'

Table 187. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIAMO_NET_TIME_MAX	730	X'000002DA'
MQIAMO_NET_TIME_MIN	731	X'000002DB'
MQIAMO_OBJECT_COUNT	732	X'000002DC'
MQIAMO_OPENS	733	X'000002DD'
MQIAMO_PUT1S	734	X'000002DE'
MQIAMO_PUTS	735	X'000002DF'
MQIAMO_PUT_MAX_BYTES	736	X'000002E0'
MQIAMO_PUT_MIN_BYTES	737	X'000002E1'
MQIAMO_PUT_RETRIES	738	X'000002E2'
MQIAMO_Q_MAX_DEPTH	739	X'000002E3'
MQIAMO_Q_MIN_DEPTH	740	X'000002E4'
MQIAMO_Q_TIME_AVG	741	X'000002E5'
MQIAMO_Q_TIME_MAX	742	X'000002E6'
MQIAMO_Q_TIME_MIN	743	X'000002E7'
MQIAMO_SETS	744	X'000002E8'
MQIAMO_CONNS_FAILED	749	X'000002ED'
MQIAMO_OPENS_FAILED	751	X'000002EF'
MQIAMO_INQS_FAILED	752	X'000002F0'
MQIAMO_SETS_FAILED	753	X'000002F1'
MQIAMO_PUTS_FAILED	754	X'000002F2'
MQIAMO_PUT1S_FAILED	755	X'000002F3'
MQIAMO_CLOSES_FAILED	757	X'000002F5'
MQIAMO_MSGS_EXPIRED	758	X'000002F6'
MQIAMO_MSGS_NOT_QUEUED	759	X'000002F7'
MQIAMO_MSGS_PURGED	760	X'000002F8'
MQIAMO_SUBS_DUR	764	X'000002FC'
MQIAMO_SUBS_NDUR	765	X'000002FD'
MQIAMO_SUBS_FAILED	766	X'000002FE'
MQIAMO_SUBRQS	767	X'000002FF'
MQIAMO_SUBRQS_FAILED	768	X'00000300'
MQIAMO_CBS	769	X'00000301'
MQIAMO_CBS_FAILED	770	X'00000302'
MQIAMO_CTLs	771	X'00000303'
MQIAMO_CTLs_FAILED	772	X'00000304'
MQIAMO_STATS	773	X'00000305'
MQIAMO_STATS_FAILED	774	X'00000306'
MQIAMO_SUB_DUR_HIGHWATER	775	X'00000307'
MQIAMO_SUB_DUR_LOWWATER	776	X'00000308'
MQIAMO_SUB_NDUR_HIGHWATER	777	X'00000309'

Table 187. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIAMO_SUB_NDUR_LOWWATER	778	X'0000030A'
MQIAMO_TOPIC_PUTS	779	X'0000030B'
MQIAMO_TOPIC_PUTS_FAILED	780	X'0000030C'
MQIAMO_TOPIC_PUT1S	781	X'0000030D'
MQIAMO_TOPIC_PUT1S_FAILED	782	X'0000030E'
MQIAMO_PUBLISH_MSG_COUNT	784	X'00000310'
MQIAMO_UNSUBS_DUR	786	X'00000312'
MQIAMO_UNSUBS_NDUR	787	X'00000313'
MQIAMO_UNSUBS_FAILED	788	X'00000314'
MQIAMO_INTERVAL	789	X'00000315'
MQIAMO_MSGS_SENT	790	X'00000316'
MQIAMO_BYTES_SENT	791	X'00000317'
MQIAMO_REPAIR_BYTES	792	X'00000318'
MQIAMO_FEEDBACK_MODE	793	X'00000319'
MQIAMO_RELIABILITY_TYPE	794	X'0000031A'
MQIAMO_LATE_JOIN_MARK	795	X'0000031B'
MQIAMO_NACKS_RCVD	796	X'0000031C'
MQIAMO_REPAIR_PKTS	797	X'0000031D'
MQIAMO_HISTORY_PKTS	798	X'0000031E'
MQIAMO_PENDING_PKTS	799	X'0000031F'
MQIAMO_PKT_RATE	800	X'00000320'
MQIAMO_MCAST_XMIT_RATE	801	X'00000321'
MQIAMO_MCAST_BATCH_TIME	802	X'00000322'
MQIAMO_MCAST_HEARTBEAT	803	X'00000323'
MQIAMO_DEST_DATA_PORT	804	X'00000324'
MQIAMO_DEST_REPAIR_PORT	805	X'00000325'
MQIAMO_ACKS_RCVD	806	X'00000326'
MQIAMO_ACTIVE_ACKERS	807	X'00000327'
MQIAMO_PKTS_SENT	808	X'00000328'
MQIAMO_TOTAL_REPAIR_PKTS	809	X'00000329'
MQIAMO_TOTAL_PKTS_SENT	810	X'0000032A'
MQIAMO_TOTAL_MSGS_SENT	811	X'0000032B'
MQIAMO_TOTAL_BYTES_SENT	812	X'0000032C'
MQIAMO_NUM_STREAMS	813	X'0000032D'
MQIAMO_ACK_FEEDBACK	814	X'0000032E'
MQIAMO_NACK_FEEDBACK	815	X'0000032F'
MQIAMO_PKTS_LOST	816	X'00000330'
MQIAMO_MSGS_RCVD	817	X'00000331'
MQIAMO_MSG_BYTES_RCVD	818	X'00000332'

Table 187. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIAMO_MSGS_DELIVERED	819	X'00000333'
MQIAMO_PKTS_PROCESSED	820	X'00000334'
MQIAMO_PKTS_DLVD	821	X'00000335'
MQIAMO_PKTS_DROPPED	822	X'00000336'
MQIAMO_PKTS_DUPLICATED	823	X'00000337'
MQIAMO_NACKS_CREATED	824	X'00000338'
MQIAMO_NACK_PKTS_SENT	825	X'00000339'
MQIAMO_REPAIR_PKTS_RQSTD	826	X'0000033A'
MQIAMO_REPAIR_PKTS_RCVD	827	X'0000033B'
MQIAMO_PKTS_REPAIRED	828	X'0000033C'
MQIAMO_TOTAL_MSGS_RCVD	829	X'0000033D'
MQIAMO_TOTAL_MSGS_BYTES_RCVD	830	X'0000033E'
MQIAMO_TOTAL_REPAIR_PKTS_RCVD	831	X'0000033F'
MQIAMO_TOTAL_REPAIR_PKTS_RQSTD	832	X'00000340'
MQIAMO_TOTAL_MSGS_PROCESSED	833	X'00000341'
MQIAMO_TOTAL_MSGS_SELECTED	834	X'00000342'
MQIAMO_TOTAL_MSGS_EXPIRED	835	X'00000343'
MQIAMO_TOTAL_MSGS_DELIVERED	836	X'00000344'
MQIAMO_TOTAL_MSGS_RETURNED	837	X'00000345'
MQIAMO_LAST_USED	837	X'00000345'

MQIAMO64_* (Command format 64-bit Integer Monitoring Parameter Types)

Table 188. Values of constants		
Name	Decimal value	Hexadecimal value
MQIAMO64_AVG_Q_TIME	703	X'000002BF'
MQIAMO64_Q_TIME_AVG	741	X'000002E5'
MQIAMO64_Q_TIME_MAX	742	X'000002E6'
MQIAMO64_Q_TIME_MIN	743	X'000002E7'
MQIAMO64_BROWSE_BYTES	745	X'000002E9'
MQIAMO64_BYTES	746	X'000002EA'
MQIAMO64_GET_BYTES	747	X'000002EB'
MQIAMO64_PUT_BYTES	748	X'000002EC'
MQIAMO64_TOPIC_PUT_BYTES	783	X'0000030F'
MQIAMO64_PUBLISH_MSG_BYTES	785	X'00000311'

MQIASY_* (Integer System Selectors)

Table 189. Values of constants		
Name	Decimal value	Hexadecimal value
MQIASY_FIRST	-1	X'FFFFFFFF'
MQIASY_CODED_CHAR_SET_ID	-1	X'FFFFFFFF'

Table 189. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQIASY_TYPE	-2	X'FFFFFFFFE'
MQIASY_COMMAND	-3	X'FFFFFFFFD'
MQIASY_MSG_SEQ_NUMBER	-4	X'FFFFFFFFC'
MQIASY_CONTROL	-5	X'FFFFFFFFB'
MQIASY_COMP_CODE	-6	X'FFFFFFFFA'
MQIASY_REASON	-7	X'FFFFFFFF9'
MQIASY_BAG_OPTIONS	-8	X'FFFFFFFF8'
MQIASY_VERSION	-9	X'FFFFFFFF7'
MQIASY_LAST_USED	-9	X'FFFFFFFF7'
MQIASY_LAST	-2000	X'FFFFF830'

MQIAUT_* (IMS information header Authenticator)

Table 190. Constant names and values	
Name	Value
MQIAUT_NONE	"␣␣␣␣␣␣␣␣"
MQIAUT_NONE_ARRAY	'␣','␣','␣','␣','␣','␣','␣','␣','␣'

Note: The symbol ␣ represents a single blank character.

MQIAV_* (Integer Attribute Values)

Table 191. Values of constants		
Name	Decimal value	Hexadecimal value
MQIAV_NOT_APPLICABLE	-1	X'FFFFFFFF'
MQIAV_UNDEFINED	-2	X'FFFFFFFFE'

MQICM_* (IMS information header Commit Modes)

Table 192. Constant names and values	
Name	Value
MQICM_COMMIT_THEN_SEND	'0'
MQICM_SEND_THEN_COMMIT	'1'

MQIDO_* (Command format Indoubt Options)

Table 193. Values of constants		
Name	Decimal value	Hexadecimal value
MQIDO_COMMIT	1	X'00000001'
MQIDO_BACKOUT	2	X'00000002'

MQIEP_* (Interface entry points)

Connection security parameters structure

Table 194. Structures of constants	
Name	Structure
MQIEP_STRUC_ID	"IEP↵"
MQIEP_STRUC_ID_ARRAY	'I','E','P','↵'

Note: The symbol ↵ represents a single blank character.

Table 195. Values of constants		
Name	Decimal value	Hexadecimal value
MQIEP_VERSION_1	1	X'00000001'
MQDXP_CURRENT_VERSION	1	X'00000001'

MQIGQ_* (Intra-Group queuing)

Table 196. Values of constants		
Name	Decimal value	Hexadecimal value
MQIGQ_DISABLED	0	X'00000000'
MQIGQ_ENABLED	1	X'00000001'

MQIGQPA_* (Intra-Group queuing Put Authority)

Table 197. Values of constants		
Name	Decimal value	Hexadecimal value
MQIGQPA_DEFAULT	1	X'00000001'
MQIGQPA_CONTEXT	2	X'00000002'
MQIGQPA_ONLY_IGQ	3	X'00000003'
MQIGQPA_ALTERNATE_OR_IGQ	4	X'00000004'

MQIIH_* (IMS information header structure and Flags)

IMS information header structure

Table 198. Structures of constants	
Name	Structure
MQIIH_STRUC_ID	"IIH↵"
MQIIH_STRUC_ID_ARRAY	'I','I','H','↵'

Note: The symbol ↵ represents a single blank character.

Table 199. Values of constants		
Name	Decimal value	Hexadecimal value
MQIIH_VERSION_1	1	X'00000001'
MQIIH_CURRENT_VERSION	1	X'00000001'
MQIIH_LENGTH_1	84	X'00000054'

IMS information header Flags

Table 200. Values of constants		
Name	Decimal value	Hexadecimal value
MQIIH_NONE	0	X'00000000'
MQIIH_PASS_EXPIRATION	1	X'00000001'
MQIIH_UNLIMITED_EXPIRATION	0	X'00000000'
MQIIH_REPLY_FORMAT_NONE	8	X'00000008'
MQIIH_IGNORE_PURG	16	X'00000010'
MQIIH_CM0_REQUEST_RESPONSE	32	X'00000020'

MQIMPO_* (Inquire message property options and structure)

Inquire message property options structure

Table 201. Structures of constants	
Name	Structure
MQIMPO_STRUC_ID	"IMPO"
MQIMPO_STRUC_ID_ARRAY	'I', 'M', 'P', 'O'

Note: The symbol ▯ represents a single blank character.

Table 202. Values of constants		
Name	Decimal value	Hexadecimal value
MQIMPO_VERSION_1	1	X'00000001'
MQIMPO_CURRENT_VERSION	1	X'00000001'

Inquire Message Property Options

Table 203. Values of constants		
Name	Decimal value	Hexadecimal value
MQIMPO_CONVERT_TYPE	2	X'00000002'
MQIMPO_QUERY_LENGTH	4	X'00000004'
MQIMPO_INQ_FIRST	0	X'00000000'
MQIMPO_INQ_NEXT	8	X'00000008'
MQIMPO_INQ_PROP_UNDER_CURSOR	16	X'00000010'
MQIMPO_CONVERT_VALUE	32	X'00000020'
MQIMPO_NONE	0	X'00000000'

MQINBD_* (Command format Inbound Dispositions)

Table 204. Values of constants		
Name	Decimal value	Hexadecimal value
MQINBD_Q_MGR	0	X'00000000'
MQINBD_GROUP	3	X'00000003'

MQIND_* (Special Index Values)

Table 205. Values of constants		
Name	Decimal value	Hexadecimal value
MQIND_NONE	-1	X'FFFFFFFF'
MQIND_ALL	-2	X'FFFFFFFE'

MQIPADDR_* (IP Address Versions)

Table 206. Values of constants		
Name	Decimal value	Hexadecimal value
MQIPADDR_IPv4	0	X'00000000'
MQIPADDR_IPv6	1	X'00000001'

MQISS_* (IMS information header Security Scopes)

Table 207. Constant names and values	
Name	Value
MQISS_CHECK	'C'
MQISS_FULL	'F'

MQIT_* (Index Types)

Table 208. Values of constants		
Name	Decimal value	Hexadecimal value
MQIT_NONE	0	X'00000000'
MQIT_MSG_ID	1	X'00000001'
MQIT_CORREL_ID	2	X'00000002'
MQIT_MSG_TOKEN	4	X'00000004'
MQIT_GROUP_ID	5	X'00000005'

MQITEM_* (Item Type for mqInquireItemInfo)

Table 209. Values of constants		
Name	Decimal value	Hexadecimal value
MQITEM_INTEGER	1	X'00000001'
MQITEM_STRING	2	X'00000002'
MQITEM_BAG	3	X'00000003'
MQITEM_BYTE_STRING	4	X'00000004'
MQITEM_INTEGER_FILTER	5	X'00000005'
MQITEM_STRING_FILTER	6	X'00000006'
MQITEM_INTEGER64	7	X'00000007'
MQITEM_BYTE_STRING_FILTER	8	X'00000008'

MQITII_* (IMS information header Transaction Instance Identifier)

Table 210. Constant names and values	
Name	Value
MQITII_NONE	X'00...00' (16 nulls)
MQITII_NONE_ARRAY	'\0', '\0', ... (16 nulls)

MQITS_* (IMS information header Transaction States)

Table 211. Constant names and values	
Name	Value
MQITS_IN_CONVERSATION	'C'
MQITS_NOT_IN_CONVERSATION	'¬'
MQITS_ARCHITECTED	'A'

Note: The symbol ¬ represents a single blank character.

MQKAI_* (KeepAlive Interval)

Table 212. Values of constants		
Name	Decimal value	Hexadecimal value
MQKAI_AUTO	-1	X'FFFFFFFF'

MQMASTER_* (Master administration)

Table 213. Values of constants		
Name	Decimal value	Hexadecimal value
MQMASTER_NO	0	X'00000000'
MQMASTER_YES	1	X'00000001'

MQMCAS_* (Command format Message Channel Agent Status)

Table 214. Values of constants		
Name	Decimal value	Hexadecimal value
MQMCAS_STOPPED	0	X'00000000'
MQMCAS_RUNNING	3	X'00000003'

MQMCAT_* (MCA Types)

Table 215. Values of constants		
Name	Decimal value	Hexadecimal value
MQMCAT_PROCESS	1	X'00000001'
MQMCAT_THREAD	2	X'00000002'

MQMCD_* (Publish/Subscribe Options Tag Information)

Publish/Subscribe Options Tag Message Content Descriptor (mcd) Tags

Table 216. Values of constants		
Name	Decimal value	Hexadecimal value
MQMCD_FOLDER_VERSION	1	X'00000001'

Publish/Subscribe Options Tag Tag names

Table 217. Constant names and values	
Name	Value
MQMCD_MSG_DOMAIN	"Msd"
MQMCD_MSG_SET	"Set"
MQMCD_MSG_TYPE	"Type"
MQMCD_MSG_FORMAT	"Fmt"

Publish/Subscribe Options Tag XML tag names

Table 218. Constant names and values	
Name	Value
MQMCD_MSG_DOMAIN_B	"<Msd>"
MQMCD_MSG_DOMAIN_E	"</Msd>"
MQMCD_MSG_SET_B	"<Set>"
MQMCD_MSG_SET_E	"</Set>"
MQMCD_MSG_TYPE_B	"<Type>"
MQMCD_MSG_TYPE_E	"</Type>"
MQMCD_MSG_FORMAT_B	"<Fmt>"
MQMCD_MSG_FORMAT_E	"</Fmt>"

Publish/Subscribe Options Tag Tag values

Table 219. Constant names and values	
Name	Value
MQMCD_DOMAIN_NONE	"none"
MQMCD_DOMAIN_NEON	"neon"
MQMCD_DOMAIN_MRM	"mrm"
MQMCD_DOMAIN_JMS_NONE	"jms_none"
MQMCD_DOMAIN_JMS_TEXT	"jms_text"
MQMCD_DOMAIN_JMS_OBJECT	"jms_object"
MQMCD_DOMAIN_JMS_MAP	"jms_map"
MQMCD_DOMAIN_JMS_STREAM	"jms_stream"
MQMCD_DOMAIN_JMS_BYTES	"jms_bytes"

MQMD_* (Message descriptor structure)

Table 220. Structures of constants	
Name	Structure
MQMD_STRUC_ID	"MD↵"
MQMD_STRUC_ID_ARRAY	'M', 'D', '↵', '↵'

Note: The symbol ↵ represents a single blank character.

Table 221. Values of constants		
Name	Decimal value	Hexadecimal value
MQMD_VERSION_1	1	X'00000001'
MQMD_VERSION_2	2	X'00000002'
MQMD_CURRENT_VERSION	2	X'00000002'

MQMDE_* (Message descriptor extension structure)

Table 222. Structures of constants	
Name	Structure
MQMDE_STRUC_ID	"MDE↵"
MQMDE_STRUC_ID_ARRAY	'M', 'D', 'E', '↵'

Note: The symbol ↵ represents a single blank character.

Table 223. Values of constants		
Name	Decimal value	Hexadecimal value
MQMDE_VERSION_2	2	X'00000002'
MQMDE_CURRENT_VERSION	2	X'00000002'
MQMDE_LENGTH_2	72	X'00000048'

MQMDEF_* (Message descriptor extension Flags)

Table 224. Values of constants		
Name	Decimal value	Hexadecimal value
MQMDEF_NONE	0	X'00000000'

MQMDS_* (Message Delivery Sequence)

Table 225. Values of constants		
Name	Decimal value	Hexadecimal value
MQMDS_PRIORITY	0	X'00000000'
MQMDS_FIFO	1	X'00000001'

MQMF_* (Message Flags)

Table 226. Values of constants		
Name	Decimal value	Hexadecimal value
MQMF_SEGMENTATION_INHIBITED	0	X'00000000'
MQMF_SEGMENTATION_ALLOWED	1	X'00000001'

Table 226. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQMF_MSG_IN_GROUP	8	X'00000008'
MQMF_LAST_MSG_IN_GROUP	16	X'00000010'
MQMF_SEGMENT	2	X'00000002'
MQMF_LAST_SEGMENT	4	X'00000004'
MQMF_NONE	0	X'00000000'

MQMHBO_* (Message handle to buffer options and structure)

Message handle to buffer options structure

Table 227. Structures of constants	
Name	Structure
MQMHBO_STRUC_ID	"MHBO"
MQMHBO_STRUC_ID_ARRAY	'M', 'H', 'B', 'O'

Note: The symbol ␣ represents a single blank character.

Table 228. Values of constants		
Name	Decimal value	Hexadecimal value
MQMHBO_VERSION_1	1	X'00000001'
MQMHBO_CURRENT_VERSION	1	X'00000001'

Message Handle To Buffer Options

Table 229. Values of constants		
Name	Decimal value	Hexadecimal value
MQMHBO_PROPERTIES_IN_MQRFH2	1	X'00000001'
MQMHBO_DELETE_PROPERTIES	2	X'00000002'
MQMHBO_NONE	0	X'00000000'

MQMI_* (Message Identifier)

Table 230. Constant names and values	
Name	Value
MQMI_NONE	X'00...00' (24 nulls)
MQMI_NONE_ARRAY	'\0', '\0', ... (24 nulls)

MQMMBI_* (Message Mark-Browse Interval)

Table 231. Values of constants		
Name	Decimal value	Hexadecimal value
MQMMBI_UNLIMITED	-1	X'FFFFFFFF'

MQMO_* (Match Options)

Table 232. Values of constants		
Name	Decimal value	Hexadecimal value
MQMO_MATCH_MSG_ID	1	X'00000001'
MQMO_MATCH_CORREL_ID	2	X'00000002'
MQMO_MATCH_GROUP_ID	4	X'00000004'
MQMO_MATCH_MSG_SEQ_NUMBER	8	X'00000008'
MQMO_MATCH_OFFSET	16	X'00000010'
MQMO_MATCH_MSG_TOKEN	32	X'00000020'
MQMO_NONE	0	X'00000000'

MQMODE_* (Command format Mode Options)

Table 233. Values of constants		
Name	Decimal value	Hexadecimal value
MQMODE_FORCE	0	X'00000000'
MQMODE_QUIESCE	1	X'00000001'
MQMODE_TERMINATE	2	X'00000002'

MQMON_* (Monitoring Values)

Table 234. Values of constants		
Name	Decimal value	Hexadecimal value
MQMON_NOT_AVAILABLE	-1	X'FFFFFFFF'
MQMON_NONE	-1	X'FFFFFFFF'
MQMON_Q_MGR	-3	X'FFFFFFFD'
MQMON_OFF	0	X'00000000'
MQMON_ON	1	X'00000001'
MQMON_DISABLED	0	X'00000000'
MQMON_ENABLED	1	X'00000001'
MQMON_LOW	17	X'00000011'
MQMON_MEDIUM	33	X'00000021'
MQMON_HIGH	65	X'00000041'

MQMT_* (Message Types)

Table 235. Values of constants		
Name	Decimal value	Hexadecimal value
MQMT_SYSTEM_FIRST	1	X'00000001'
MQMT_REQUEST	1	X'00000001'
MQMT_REPLY	2	X'00000002'
MQMT_DATAGRAM	8	X'00000008'
MQMT_REPORT	4	X'00000004'
MQMT_MQE_FIELDS_FROM_MQE	112	X'00000070'

Table 235. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQMT_MQE_FIELDS	113	X'00000071'
MQMT_SYSTEM_LAST	65535	X'0000FFFF'
MQMT_APPL_FIRST	65536	X'00010000'
MQMT_APPL_LAST	99999999	X'3B9AC9FF'

MQMTOK_* (Message Token)

Table 236. Constant names and values	
Name	Value
MQMTOK_NONE	X'00...00' (16 nulls)
MQMTOK_NONE_ARRAY	'\0','\0',... (16 nulls)

Table 237. Values of constants		
Name	Decimal value	Hexadecimal value
MQMTOK_NONE	X'00...00'	(16 nulls)
MQMTOK_NONE_ARRAY	'\0','\0',...	(16 nulls)

MQNC_* (Name Count)

Table 238. Values of constants		
Name	Decimal value	Hexadecimal value
MQNC_MAX_NAMELIST_NAME_COUNT	256	X'00000100'

MQNPM_* (Nonpersistent Message Class)

Table 239. Values of constants		
Name	Decimal value	Hexadecimal value
MQNPM_CLASS_NORMAL	0	X'00000000'
MQNPM_CLASS_HIGH	10	X'0000000A'

MQNPMS_* (NonPersistent-Message Speeds)

Table 240. Values of constants		
Name	Decimal value	Hexadecimal value
MQNPMS_NORMAL	1	X'00000001'
MQNPMS_FAST	2	X'00000002'

MQNT_* (Namelist Types)

Table 241. Values of constants		
Name	Decimal value	Hexadecimal value
MQNT_NONE	0	X'00000000'
MQNT_Q	1	X'00000001'
MQNT_CLUSTER	2	X'00000002'
MQNT_AUTH_INFO	4	X'00000004'

Table 241. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQNT_ALL	1001	X'000003E9'

MQNVS_* (Names for Name/Value String)

Table 242. Constant names and values	
Name	Value
MQNVS_APPL_TYPE	"OPT_APP_GRP"
MQNVS_MSG_TYPE	"OPT_MSG_TYPE"

Note: The symbol ~ represents a single blank character.

MQOA_* (Limits for Selectors for Object Attributes)

Table 243. Values of constants		
Name	Decimal value	Hexadecimal value
MQOA_FIRST	1	X'00000001'
MQOA_LAST	9000	X'00002328'

MQOD_* (Object descriptor structure)

Table 244. Structures of constants	
Name	Structure
MQOD_STRUC_ID	"OD~"
MQOD_STRUC_ID_ARRAY	'0', 'D', '~', '~'

Note: The symbol ~ represents a single blank character.

Table 245. Values of constants		
Name	Decimal value	Hexadecimal value
MQOD_VERSION_1	1	X'00000001'
MQOD_VERSION_2	2	X'00000002'
MQOD_VERSION_3	3	X'00000003'
MQOD_VERSION_4	4	X'00000004'
MQOD_CURRENT_VERSION	4	X'00000004'
MQOD_CURRENT_LENGTH	(value differs by platform or version)	

MQOII_* (Object Instance Identifier)

Table 246. Constant names and values	
Name	Value
MQOII_NONE	X'00...00' (24 nulls)
MQOII_NONE_ARRAY	'\0', '\0', ... (24 nulls)

MQOL_* (Original Length)

Table 247. Values of constants		
Name	Decimal value	Hexadecimal value
MQOL_UNDEFINED	-1	X'FFFFFFFF'

MQOM_* (Obsolete Db2 Messages options on Inquire Group)

Table 248. Values of constants		
Name	Decimal value	Hexadecimal value
MQOM_NO	0	X'00000000'
MQOM_YES	1	X'00000001'

MQOO_* (Open Options)

Table 249. Values of constants		
Name	Decimal value	Hexadecimal value
MQOO_BIND_AS_Q_DEF	0	X'00000000'
MQOO_READ_AHEAD_AS_Q_DEF	0	X'00000000'
MQOO_INPUT_AS_Q_DEF	1	X'00000001'
MQOO_INPUT_SHARED	2	X'00000002'
MQOO_INPUT_EXCLUSIVE	4	X'00000004'
MQOO_BROWSE	8	X'00000008'
MQOO_OUTPUT	16	X'00000010'
MQOO_INQUIRE	32	X'00000020'
MQOO_SET	64	X'00000040'
MQOO_SAVE_ALL_CONTEXT	128	X'00000080'
MQOO_PASS_IDENTITY_CONTEXT	256	X'00000100'
MQOO_PASS_ALL_CONTEXT	512	X'00000200'
MQOO_SET_IDENTITY_CONTEXT	1024	X'00000400'
MQOO_SET_ALL_CONTEXT	2048	X'00000800'
MQOO_ALTERNATE_USER_AUTHORITY	4096	X'00001000'
MQOO_FAIL_IF QUIESCING	8192	X'00002000'
MQOO_BIND_ON_OPEN	16384	X'00004000'
MQOO_BIND_NOT_FIXED	32768	X'00008000'
MQOO_CO_OP	131072	X'00020000'
MQOO_RESOLVE_LOCAL_TOPIC	262144	X'00040000'
MQOO_NO_READ_AHEAD	524288	X'00080000'
MQOO_READ_AHEAD	1048576	X'00100000'
MQOO_BIND_ON_GROUP	4194304	X'00400000'

MQOO_* (Following used in C++ only)

Table 250. Values of constants		
Name	Decimal value	Hexadecimal value
MQOO_RESOLVE_NAMES	65536	X'00010000'
MQOO_RESOLVE_LOCAL_Q	262144	X'00040000'

MQOP_* (Operation codes for MQCTL and MQCB)

Operation codes for MQCTL

Table 251. Values of constants		
Name	Decimal value	Hexadecimal value
MQOP_START	1	X'00000001'
MQOP_START_WAIT	2	X'00000002'
MQOP_STOP	4	X'00000004'

Operation codes for MQCB

Table 252. Values of constants		
Name	Decimal value	Hexadecimal value
MQOP_REGISTER	256	X'00000100'
MQOP_DEREGISTER	512	X'00000200'

Operation codes for MQCTL and MQCB

Table 253. Values of constants		
Name	Decimal value	Hexadecimal value
MQOP_SUSPEND	65536	X'00010000'
MQOP_RESUME	131072	X'00020000'

MQOPEN_* (Values related to MQOPEN_PRIV structure)

Table 254. Values of constants		
Name	Decimal value	Hexadecimal value
MQOPEN_PRIV_VERSION_1	1	X'00000001'
MQOPEN_PRIV_CURRENT_VERSION	1	X'00000001'

MQOPER_* (Activity Operations)

Table 255. Values of constants		
Name	Decimal value	Hexadecimal value
MQOPER_SYSTEM_FIRST	0	X'00000000'
MQOPER_UNKNOWN	0	X'00000000'
MQOPER_BROWSE	1	X'00000001'
MQOPER_DISCARD	2	X'00000002'
MQOPER_GET	3	X'00000003'

Table 255. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQOPER_PUT	4	X'00000004'
MQOPER_PUT_REPLY	5	X'00000005'
MQOPER_PUT_REPORT	6	X'00000006'
MQOPER_RECEIVE	7	X'00000007'
MQOPER_SEND	8	X'00000008'
MQOPER_TRANSFORM	9	X'00000009'
MQOPER_PUBLISH	10	X'0000000A'
MQOPER_EXCLUDED_PUBLISH	11	X'0000000B'
MQOPER_DISCARDED_PUBLISH	12	X'0000000C'
MQOPER_SYSTEM_LAST	65535	X'000FFFF'
MQOPER_APPL_FIRST	65536	X'00010000'
MQOPER_APPL_LAST	99999999	X'3B9AC9FF'


MQOT_* (Object Types and Extended Object Types)

Object Types

Table 256. Values of constants		
Name	Decimal value	Hexadecimal value
MQOT_NONE	0	X'00000000'
MQOT_Q	1	X'00000001'
MQOT_NAMELIST	2	X'00000002'
MQOT_PROCESS	3	X'00000003'
MQOT_STORAGE_CLASS	4	X'00000004'
MQOT_Q_MGR	5	X'00000005'
MQOT_CHANNEL	6	X'00000006'
MQOT_AUTH_INFO	7	X'00000007'
MQOT_TOPIC	8	X'00000008'
MQOT_CF_STRUC	10	X'0000000A'
MQOT_LISTENER	11	X'0000000B'
MQOT_SERVICE	12	X'0000000C'
MQOT_RESERVED_1	999	X'000003E7'

Extended Object Types

Table 257. Values of constants		
Name	Decimal value	Hexadecimal value
MQOT_ALL	1001	X'000003E9'
MQOT_ALIAS_Q	1002	X'000003EA'
MQOT_MODEL_Q	1003	X'000003EB'
MQOT_LOCAL_Q	1004	X'000003EC'
MQOT_REMOTE_Q	1005	X'000003ED'

Table 257. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQOT_SENDER_CHANNEL	1007	X'000003EF'
MQOT_SERVER_CHANNEL	1008	X'000003F0'
MQOT_REQUESTER_CHANNEL	1009	X'000003F1'
MQOT_RECEIVER_CHANNEL	1010	X'000003F2'
MQOT_CURRENT_CHANNEL	1011	X'000003F3'
MQOT_SAVED_CHANNEL	1012	X'000003F4'
MQOT_SVRCONN_CHANNEL	1013	X'000003F5'
MQOT_CLNTCONN_CHANNEL	1014	X'000003F6'
MQOT_SHORT_CHANNEL	1015	X'000003F7'
MQOT_CHLAUTH	1016	X'000003F8'
MQOT_REMOTE_Q_MGR_NAME	1017	X'000003F9'
MQOT_PROT_POLICY	1019	X'000003FB'
MQOT_TT_CHANNEL	1020	X'000003FC'
MQOT_AMQP_CHANNEL	1021	X'000003FD'
 MQOT_AUTH_REC	1022	X'000003FE'

MQPA_* (Put Authority)

Table 258. Values of constants		
Name	Decimal value	Hexadecimal value
MQPA_DEFAULT	1	X'00000001'
MQPA_CONTEXT	2	X'00000002'
MQPA_ONLY_MCA	3	X'00000003'
MQPA_ALTERNATE_OR_MCA	4	X'00000004'

MQPD_* (Property descriptor, support and context)

Property descriptor structure

Table 259. Structures of constants	
Name	Structure
MQPD_STRUC_ID	"PD↵"
MQPD_STRUC_ID_ARRAY	'P','D','↵','↵'

Note: The symbol ↵ represents a single blank character.

Table 260. Values of constants		
Name	Decimal value	Hexadecimal value
MQPD_VERSION_1	1	X'00000001'
MQPD_CURRENT_VERSION	1	X'00000001'

Note: The symbol ↵ represents a single blank character.

Property Descriptor Options

Table 261. Values of constants		
Name	Decimal value	Hexadecimal value
MQPD_NONE	0	X'00000000'

Property Support Options

Table 262. Values of constants		
Name	Decimal value	Hexadecimal value
MQPD_SUPPORT_OPTIONAL	1	X'00000001'
MQPD_SUPPORT_REQUIRED	1048576	X'00100000'
MQPD_SUPPORT_REQUIRED_IF_LOCAL	1024	X'00000400'
MQPD_REJECT_UNSUP_MASK	-1048576	X'FFF00000'
MQPD_ACCEPT_UNSUP_IF_XMIT_MASK	1047552	X'000FFC00'
MQPD_ACCEPT_UNSUP_MASK	1023	X'000003FF'

Property Context


Table 263. Values of constants		
Name	Decimal value	Hexadecimal value
MQPD_NO_CONTEXT	0	X'00000000'
MQPD_USER_CONTEXT	1	X'00000001'

MQPER_* (Persistence Values)

Table 264. Values of constants		
Name	Decimal value	Hexadecimal value
MQPER_PERSISTENCE_AS_PARENT	-1	X'FFFFFFFF'
MQPER_NOT_PERSISTENT	0	X'00000000'
MQPER_PERSISTENT	1	X'00000001'
MQPER_PERSISTENCE_AS_Q_DEF	2	X'00000002'
MQPER_PERSISTENCE_AS_TOPIC_DEF	2	X'00000002'

MQPL_* (Platforms)

Table 265. Values of constants		
Name	Decimal value	Hexadecimal value
MQPL_MVS	1	X'00000001'
MQPL_OS390	1	X'00000001'
MQPL_ZOS	1	X'00000001'
MQPL_OS2	2	X'00000002'
MQPL_AIX	3	X'00000003'
MQPL_UNIX	3	X'00000003'
MQPL_OS400	4	X'00000004'
MQPL_WINDOWS	5	X'00000005'

Table 265. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQPL_WINDOWS_NT	11	X'0000000B'
MQPL_VMS	12	X'0000000C'
MQPL_NSK	13	X'0000000D'
MQPL_NSS	13	X'0000000D'
MQPL_OPEN_TP1	15	X'0000000F'
MQPL_VM	18	X'00000012'
MQPL_TPF	23	X'00000017'
MQPL_VSE	27	X'0000001B'
 MQPL_APPLIANCE	28	X'0000001C'
MQPL_NATIVE	1	X'00000001'

MQPMO_* (Put message options and structure for publish mask)

Put message options structure

Table 266. Structures of constants	
Name	Structure
MQPMO_STRUC_ID	"PMO¬"
MQPMO_STRUC_ID_ARRAY	'P','M','O','¬'

Note: The symbol ¬ represents a single blank character.

Table 267. Values of constants		
Name	Decimal value	Hexadecimal value
MQPMO_VERSION_1	1	X'00000001'
MQPMO_VERSION_2	2	X'00000002'
MQPMO_VERSION_3	3	X'00000003'
MQPMO_CURRENT_VERSION	3	X'00000003'
MQPMO_CURRENT_LENGTH	(value differs by platform or version)	

Put Message Options

Table 268. Values of constants		
Name	Decimal value	Hexadecimal value
MQPMO_SYNCPOINT	2	X'00000002'
MQPMO_NO_SYNCPOINT	4	X'00000004'
MQPMO_DEFAULT_CONTEXT	32	X'00000020'
MQPMO_NEW_MSG_ID	64	X'00000040'
MQPMO_NEW_CORREL_ID	128	X'00000080'
MQPMO_PASS_IDENTITY_CONTEXT	256	X'00000100'
MQPMO_PASS_ALL_CONTEXT	512	X'00000200'
MQPMO_SET_IDENTITY_CONTEXT	1024	X'00000400'

Table 268. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQPMO_SET_ALL_CONTEXT	2048	X'00000800'
MQPMO_ALTERNATE_USER_AUTHORITY	4096	X'00001000'
MQPMO_FAIL_IF QUIESCING	8192	X'00002000'
MQPMO_NO_CONTEXT	16384	X'00004000'
MQPMO_LOGICAL_ORDER	32768	X'00008000'
MQPMO_ASYNC_RESPONSE	65536	X'00010000'
MQPMO_SYNC_RESPONSE	131072	X'00020000'
MQPMO_RESOLVE_LOCAL_Q	262144	X'00040000'
MQPMO_RETAIN	2097152	X'00200000'
MQPMO_MD_FOR_OUTPUT_ONLY	8388608	X'00800000'
MQPMO_SCOPE_QMGR	67108864	X'04000000'
MQPMO_SUPPRESS_REPLYTO	134217728	X'08000000'
MQPMO_NOT_OWN_SUBS	268435456	X'10000000'
MQPMO_RESPONSE_AS_Q_DEF	0	X'00000000'
MQPMO_RESPONSE_AS_TOPIC_DEF	0	X'00000000'
MQPMO_NONE	0	X'00000000'

Put Message Options for publish mask

Table 269. Values of constants		
Name	Decimal value	Hexadecimal value
MQPMO_PUB_OPTIONS_MASK	2097152	X'00200000'

MQPMRF_* (Put Message Record Fields)

Table 270. Values of constants		
Name	Decimal value	Hexadecimal value
MQPMRF_MSG_ID	1	X'00000001'
MQPMRF_CORREL_ID	2	X'00000002'
MQPMRF_GROUP_ID	4	X'00000004'
MQPMRF_FEEDBACK	8	X'00000008'
MQPMRF_ACCOUNTING_TOKEN	16	X'00000010'
MQPMRF_NONE	0	X'00000000'

MQPO_* (Command format Purge Options)

Table 271. Values of constants		
Name	Decimal value	Hexadecimal value
MQPO_YES	1	X'00000001'
MQPO_NO	0	X'00000000'

MQPRI_* (Priority)

Table 272. Values of constants		
Name	Decimal value	Hexadecimal value
MQPRI_PRIORITY_AS_Q_DEF	-1	X'FFFFFFFF'
MQPRI_PRIORITY_AS_PARENT	-2	X'FFFFFFFE'
MQPRI_PRIORITY_AS_PUBLISHED	-3	X'FFFFFFFD'
MQPRI_PRIORITY_AS_TOPIC_DEF	-1	X'FFFFFFFF'

MQPROP_* (Queue and Channel Property Control Values and Maximum Properties Length)

Queue and Channel Property Control Values

Table 273. Values of constants		
Name	Decimal value	Hexadecimal value
MQPROP_COMPATIBILITY	0	X'00000000'
MQPROP_NONE	1	X'00000001'
MQPROP_ALL	2	X'00000002'
MQPROP_FORCE_MQRFH2	3	X'00000003'

Maximum Properties Length

Table 274. Values of constants		
Name	Decimal value	Hexadecimal value
MQPROP_UNRESTRICTED_LENGTH	-1	X'FFFFFFFF'

MQPRT_* (Put Response Values)

Table 275. Values of constants		
Name	Decimal value	Hexadecimal value
MQPRT_RESPONSE_AS_PARENT	0	X'00000000'
MQPRT_SYNC_RESPONSE	1	X'00000001'
MQPRT_ASYNC_RESPONSE	2	X'00000002'

MQPS_* (Publish/Subscribe)

Command format Publish/Subscribe Status

Table 276. Values of constants		
Name	Decimal value	Hexadecimal value
MQPS_STATUS_INACTIVE	0	X'00000000'
MQPS_STATUS_STARTING	1	X'00000001'
MQPS_STATUS_STOPPING	2	X'00000002'
MQPS_STATUS_ACTIVE	3	X'00000003'
MQPS_STATUS_COMPAT	4	X'00000004'
MQPS_STATUS_ERROR	5	X'00000005'

Table 276. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQPS_STATUS_REFUSED	6	X'00000006'

Publish/Subscribe Tags as strings

MQPS_COMMAND	"MQPSCommand"
MQPS_COMP_CODE	"MQPSCompCode"
MQPS_CORREL_ID	"MQPSCorrelId"
MQPS_DELETE_OPTIONS	"MQPSDelOpts"
MQPS_ERROR_ID	"MQPSErrorId"
MQPS_ERROR_POS	"MQPSErrorPos"
MQPS_INTEGER_DATA	"MQPSIntData"
MQPS_PARAMETER_ID	"MQSParmId"
MQPS_PUBLICATION_OPTIONS	"MQSPubOpts"
MQPS_PUBLISH_TIMESTAMP	"MQSPubTime"
MQPS_Q_MGR_NAME	"MQPSQMgrName"
MQPS_Q_NAME	"MQPSQName"
MQPS_REASON	"MQPSReason"
MQPS_REASON_TEXT	"MQPSReasonText"
MQPS_REGISTRATION_OPTIONS	"MQPSRegOpts"
MQPS_SEQUENCE_NUMBER	"MQPSSeqNum"
MQPS_STREAM_NAME	"MQPSStreamName"
MQPS_STRING_DATA	"MQPSStringData"
MQPS_SUBSCRIPTION_IDENTITY	"MQPSSubIdentity"
MQPS_SUBSCRIPTION_NAME	"MQPSSubName"
MQPS_SUBSCRIPTION_USER_DATA	"MQPSSubUserData"
MQPS_TOPIC	"MQPSTopic"
MQPS_USER_ID	"MQPSUserId"

Publish/Subscribe Tags as blank-enclosed strings

MQPS_COMMAND_B	"¬MQPSCommand¬"
MQPS_COMP_CODE_B	"¬MQPSCompCode¬"
MQPS_CORREL_ID_B	"¬MQPSCorrelId¬"
MQPS_DELETE_OPTIONS_B	"¬MQPSDelOpts¬"
MQPS_ERROR_ID_B	"¬MQPSErrorId¬"
MQPS_ERROR_POS_B	"¬MQPSErrorPos¬"
MQPS_INTEGER_DATA_B	"¬MQPSIntData¬"

MQPS_PARAMETER_ID_B	"¬MQPSParmId¬"
MQPS_PUBLICATION_OPTIONS_B	"¬MQPSPubOpts¬"
MQPS_PUBLISH_TIMESTAMP_B	"¬MQPSPubTime¬"
MQPS_Q_MGR_NAME_B	"¬MQPSQMgrName¬"
MQPS_Q_NAME_B	"¬MQPSQName¬"
MQPS_REASON_B	"¬MQPSReason¬"
MQPS_REASON_TEXT_B	"¬MQPSReasonText¬"
MQPS_REGISTRATION_OPTIONS_B	"¬MQPSRegOpts¬"
MQPS_SEQUENCE_NUMBER_B	"¬MQPSSeqNum¬"
MQPS_STREAM_NAME_B	"¬MQPSStreamName¬"
MQPS_STRING_DATA_B	"¬MQPSStringData¬"
MQPS_SUBSCRIPTION_IDENTITY_B	"¬MQPSSubIdentity¬"
MQPS_SUBSCRIPTION_NAME_B	"¬MQPSSubName¬"
MQPS_SUBSCRIPTION_USER_DATA_B	"¬MQPSSubUserData¬"
MQPS_TOPIC_B	"¬MQPSTopic¬"
MQPS_USER_ID_B	"¬MQPSUserId¬"

Note: The symbol ¬ represents a single blank character.

Publish/Subscribe Command Tag Values as strings

MQPS_DELETE_PUBLICATION	"DeletePub"
MQPS_DEREGISTER_PUBLISHER	"DeregPub"
MQPS_DEREGISTER_SUBSCRIBER	"DeregSub"
MQPS_PUBLISH	"Publish"
MQPS_REGISTER_PUBLISHER	"RegPub"
MQPS_REGISTER_SUBSCRIBER	"RegSub"
MQPS_REQUEST_UPDATE	"ReqUpdate"

Publish/Subscribe Command Tag Values as blank-enclosed strings

MQPS_DELETE_PUBLICATION_B	"¬DeletePub¬"
MQPS_DEREGISTER_PUBLISHER_B	"¬DeregPub¬"
MQPS_DEREGISTER_SUBSCRIBER_B	"¬DeregSub¬"
MQPS_PUBLISH_B	"¬Publish¬"
MQPS_REGISTER_PUBLISHER_B	"¬RegPub¬"
MQPS_REGISTER_SUBSCRIBER_B	"¬RegSub¬"
MQPS_REQUEST_UPDATE_B	"¬ReqUpdate¬"

Note: The symbol ¬ represents a single blank character.

Publish/Subscribe Options Tag Values as strings

MQPS_ADD_NAME	"AddName"
MQPS_ANONYMOUS	"Anon"
MQPS_CORREL_ID_AS_IDENTITY	"CorrelAsId"
MQPS_DEREGISTER_ALL	"DeregAll"
MQPS_DIRECT_REQUESTS	"DirectReq"
MQPS_DUPLICATES_OK	"DupsOK"
MQPS_FULL_RESPONSE	"FullResp"
MQPS_INCLUDE_STREAM_NAME	"InclStreamName"
MQPS_INFORM_IF_RETAINED	"InformIfRet"
MQPS_IS_RETAINED_PUBLICATION	"IsRetainedPub"
MQPS_JOIN_EXCLUSIVE	"JoinExcl"
MQPS_JOIN_SHARED	"JoinShared"
MQPS_LEAVE_ONLY	"LeaveOnly"
MQPS_LOCAL	"Local"
MQPS_LOCKED	"Locked"
MQPS_NEW_PUBLICATIONS_ONLY	"NewPubsOnly"
MQPS_NO_ALTERATION	"NoAlter"
MQPS_NO_REGISTRATION	"NoReg"
MQPS_NON_PERSISTENT	"NonPers"
MQPS_NONE	"None"
MQPS_OTHER_SUBSCRIBERS_ONLY	"OtherSubsOnly"
MQPS_PERSISTENT	"Pers"
MQPS_PERSISTENT_AS_PUBLISH	"PersAsPub"
MQPS_PERSISTENT_AS_Q	"PersAsQueue"
MQPS_PUBLISH_ON_REQUEST_ONLY	"PubOnReqOnly"
MQPS_RETAIN_PUBLICATION	"RetainPub"
MQPS_VARIABLE_USER_ID	"VariableUserId"

Publish/Subscribe Options Tag Values as blank-enclosed strings

MQPS_ADD_NAME_B	"-AddName-"
MQPS_ANONYMOUS_B	"-Anon-"
MQPS_CORREL_ID_AS_IDENTITY_B	"-CorrelAsId-"
MQPS_DEREGISTER_ALL_B	"-DeregAll-"
MQPS_DIRECT_REQUESTS_B	"-DirectReq-"
MQPS_DUPLICATES_OK_B	"-DupsOK-"

MQPS_FULL_RESPONSE_B	"¬FullResp¬"
MQPS_INCLUDE_STREAM_NAME_B	"¬InclStreamName¬"
MQPS_INFORM_IF_RETAINED_B	"¬InformIfRet¬"
MQPS_IS_RETAINED_PUBLICATION_B	"¬IsRetainedPub¬"
MQPS_JOIN_EXCLUSIVE_B	"¬JoinExcl¬"
MQPS_JOIN_SHARED_B	"¬JoinShared¬"
MQPS_LEAVE_ONLY_B	"¬LeaveOnly¬"
MQPS_LOCAL_B	"¬Local¬"
MQPS_LOCKED_B	"¬Locked¬"
MQPS_NEW_PUBLICATIONS_ONLY_B	"¬NewPubsOnly¬"
MQPS_NO_ALTERATION_B	"¬NoAlter¬"
MQPS_NO_REGISTRATION_B	"¬NoReg¬"
MQPS_NON_PERSISTENT_B	"¬NonPers¬"
MQPS_NONE_B	"¬None¬"
MQPS_OTHER_SUBSCRIBERS_ONLY_B	"¬OtherSubsOnly¬"
MQPS_PERSISTENT_B	"¬Pers¬"
MQPS_PERSISTENT_AS_PUBLISH_B	"¬PersAsPub¬"
MQPS_PERSISTENT_AS_Q_B	"¬PersAsQueue¬"
MQPS_PUBLISH_ON_REQUEST_ONLY_B	"¬PubOnReqOnly¬"
MQPS_RETAIN_PUBLICATION_B	"¬RetainPub¬"
MQPS_VARIABLE_USER_ID_B	"¬VariableUserId¬"

Note: The symbol ¬ represents a single blank character.

MQPSC_* (Publish/Subscribe Options Tag Publish/Subscribe Command Folder (psc) Tags)

Table 277. Values of constants		
Name	Decimal value	Hexadecimal value
MQPSC_FOLDER_VERSION	1	X'00000001'

MQPSC_* (Publish/Subscribe Options Tag Tag names)

MQPSC_COMMAND	"Command"
MQPSC_REGISTRATION_OPTION	"RegOpt"
MQPSC_PUBLICATION_OPTION	"PubOpt"
MQPSC_DELETE_OPTION	"DelOpt"
MQPSC_TOPIC	"Topic"
MQPSC_SUBSCRIPTION_POINT	"SubPoint"
MQPSC_FILTER	"Filter"

MQPSC_Q_MGR_NAME	"QMgrName"
MQPSC_Q_NAME	"QName"
MQPSC_PUBLISH_TIMESTAMP	"PubTime"
MQPSC_SEQUENCE_NUMBER	"SeqNum"
MQPSC_SUBSCRIPTION_NAME	"SubName"
MQPSC_SUBSCRIPTION_IDENTITY	"SubIdentity"
MQPSC_SUBSCRIPTION_USER_DATA	"SubUserData"
MQPSC_CORREL_ID	"CorrelId"

MQPSC_* (Publish/Subscribe Options Tag XML tag names)

MQPSC_COMMAND_B	"<Command>"
MQPSC_COMMAND_E	"</Command>"
MQPSC_REGISTRATION_OPTION_B	"<RegOpt>"
MQPSC_REGISTRATION_OPTION_E	"</RegOpt>"
MQPSC_PUBLICATION_OPTION_B	"<PubOpt>"
MQPSC_PUBLICATION_OPTION_E	"</PubOpt>"
MQPSC_DELETE_OPTION_B	"<DelOpt>"
MQPSC_DELETE_OPTION_E	"</DelOpt>"
MQPSC_TOPIC_B	"<Topic>"
MQPSC_TOPIC_E	"</Topic>"
MQPSC_SUBSCRIPTION_POINT_B	"<SubPoint>"
MQPSC_SUBSCRIPTION_POINT_E	"</SubPoint>"
MQPSC_FILTER_B	"<Filter>"
MQPSC_FILTER_E	"</Filter>"
MQPSC_Q_MGR_NAME_B	"<QMgrName>"
MQPSC_Q_MGR_NAME_E	"</QMgrName>"
MQPSC_Q_NAME_B	"<QName>"
MQPSC_Q_NAME_E	"</QName>"
MQPSC_PUBLISH_TIMESTAMP_B	"<PubTime>"
MQPSC_PUBLISH_TIMESTAMP_E	"</PubTime>"
MQPSC_SEQUENCE_NUMBER_B	"<SeqNum>"
MQPSC_SEQUENCE_NUMBER_E	"</SeqNum>"
MQPSC_SUBSCRIPTION_NAME_B	"<SubName>"
MQPSC_SUBSCRIPTION_NAME_E	"</SubName>"
MQPSC_SUBSCRIPTION_IDENTITY_B	"<SubIdentity>"
MQPSC_SUBSCRIPTION_IDENTITY_E	"</SubIdentity>"

MQPSC_SUBSCRIPTION_USER_DATA_B	"<SubUserData>"
MQPSC_SUBSCRIPTION_USER_DATA_E	"</SubUserData>"
MQPSC_CORREL_ID_B	"<CorrelId>"
MQPSC_CORREL_ID_E	"</CorrelId>"

MQPSC_* (Publish/Subscribe Options Tag Values as strings)

MQPSC_DELETE_PUBLICATION	"DeletePub"
MQPSC_DEREGISTER_SUBSCRIBER	"DeregSub"
MQPSC_PUBLISH	"Publish"
MQPSC_REGISTER_SUBSCRIBER	"RegSub"
MQPSC_REQUEST_UPDATE	"ReqUpdate"

MQPSC_* (Publish/Subscribe Options Tag Values as strings)

MQPSC_ADD_NAME	"AddName"
MQPSC_CORREL_ID_AS_IDENTITY	"CorrelAsId"
MQPSC_DEREGISTER_ALL	"DeregAll"
MQPSC_DUPLICATES_OK	"DupsOK"
MQPSC_FULL_RESPONSE	"FullResp"
MQPSC_INFORM_IF_RETAINED	"InformIfRet"
MQPSC_IS_RETAINED_PUB	"IsRetainedPub"
MQPSC_JOIN_SHARED	"JoinShared"
MQPSC_JOIN_EXCLUSIVE	"JoinExcl"
MQPSC_LEAVE_ONLY	"LeaveOnly"
MQPSC_LOCAL	"Local"
MQPSC_LOCKED	"Locked"
MQPSC_NEW_PUBS_ONLY	"NewPubsOnly"
MQPSC_NO_ALTERATION	"NoAlter"
MQPSC_NON_PERSISTENT	"NonPers"
MQPSC_OTHER_SUBS_ONLY	"OtherSubsOnly"
MQPSC_PERSISTENT	"Pers"
MQPSC_PERSISTENT_AS_PUBLISH	"PersAsPub"
MQPSC_PERSISTENT_AS_Q	"PersAsQueue"
MQPSC_NONE	"None"
MQPSC_PUB_ON_REQUEST_ONLY	"PubOnReqOnly"
MQPSC_RETAIN_PUB	"RetainPub"
MQPSC_VARIABLE_USER_ID	"VariableUserId"

MQPSCR_* (Publish/Subscribe Options)

Publish/Subscribe Options Tag Publish/Subscribe Response Folder (pscr) Tags

Table 278. Values of constants		
Name	Decimal value	Hexadecimal value
MQPSCR_FOLDER_VERSION	1	X'00000001'

Publish/Subscribe Options Tag Tag names

MQPSCR_COMPLETION	"Completion"
MQPSCR_RESPONSE	"Response"
MQPSCR_REASON	"Reason"

Publish/Subscribe Options Tag XML tag names

MQPSCR_COMPLETION_B	"<Completion>"
MQPSCR_COMPLETION_E	"</Completion>"
MQPSCR_RESPONSE_B	"<Response>"
MQPSCR_RESPONSE_E	"</Response>"
MQPSCR_REASON_B	"<Reason>"
MQPSCR_REASON_E	"</Reason>"

Publish/Subscribe Options Tag Tag values

MQPSCR_OK	"ok"
MQPSCR_WARNING	"warning"
MQPSCR_ERROR	"error"

MQPSM_* (Pub/Sub Mode)

Table 279. Values of constants		
Name	Decimal value	Hexadecimal value
MQPSM_DISABLED	0	X'00000000'
MQPSM_COMPAT	1	X'00000001'
MQPSM_ENABLED	2	X'00000002'

MQPSPROP_* (Pub/Sub Message Properties)

Table 280. Values of constants		
Name	Decimal value	Hexadecimal value
MQPSPROP_NONE	0	X'00000000'
MQPSPROP_COMPAT	1	X'00000001'
MQPSPROP_RFH2	2	X'00000002'
MQPSPROP_MSGPROP	3	X'00000003'

MQPSST_* (Command format Pub/Sub Status Type)

Table 281. Values of constants		
Name	Decimal value	Hexadecimal value
MQPSST_ALL	0	X'00000000'
MQPSST_LOCAL	1	X'00000001'
MQPSST_PARENT	2	X'00000002'
MQPSST_CHILD	3	X'00000003'

MQPUBO_* (Publish/Subscribe Publication Options)

Table 282. Values of constants		
Name	Decimal value	Hexadecimal value
MQPUBO_NONE	0	X'00000000'
MQPUBO_CORREL_ID_AS_IDENTITY	1	X'00000001'
MQPUBO_RETAIN_PUBLICATION	2	X'00000002'
MQPUBO_OTHER_SUBSCRIBERS_ONLY	4	X'00000004'
MQPUBO_NO_REGISTRATION	8	X'00000008'
MQPUBO_IS_RETAINED_PUBLICATION	16	X'00000010'

MQPXP_* (Publish/subscribe routing exit parameter structure)

Table 283. Structures of constants	
Name	Structure
MQPXP_STRUC_ID	"PXP↵"
MQPXP_STRUC_ID_ARRAY	'P','X','P','↵'

Note: The symbol ↵ represents a single blank character.

Table 284. Values of constants		
Name	Decimal value	Hexadecimal value
MQPXP_VERSION_1	1	X'00000001'
MQPXP_CURRENT_VERSION	1	X'00000001'

MQQA_* (Queue attributes)

Inhibit Get Values

Table 285. Values of constants		
Name	Decimal value	Hexadecimal value
MQQA_GET_INHIBITED	1	X'00000001'
MQQA_GET_ALLOWED	0	X'00000000'

Inhibit Put Values

Table 286. Values of constants		
Name	Decimal value	Hexadecimal value
MQQA_PUT_INHIBITED	1	X'00000001'

Table 286. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQQA_PUT_ALLOWED	0	X'00000000'

Queue Shareability

Table 287. Values of constants		
Name	Decimal value	Hexadecimal value
MQQA_SHAREABLE	1	X'00000001'
MQQA_NOT_SHAREABLE	0	X'00000000'

Back-Out Hardening

Table 288. Values of constants		
Name	Decimal value	Hexadecimal value
MQQA_BACKOUT_HARDENED	1	X'00000001'
MQQA_BACKOUT_NOT_HARDENED	0	X'00000000'

MQQDT_* (Queue Definition Types)

Table 289. Values of constants		
Name	Decimal value	Hexadecimal value
MQQDT_PREDEFINED	1	X'00000001'
MQQDT_PERMANENT_DYNAMIC	2	X'00000002'
MQQDT_TEMPORARY_DYNAMIC	3	X'00000003'
MQQDT_SHARED_DYNAMIC	4	X'00000004'

MQQF_* (Queue Flags)

Table 290. Values of constants		
Name	Decimal value	Hexadecimal value
MQQF_LOCAL_Q	1	X'00000001'
MQQF_CLWL_USEQ_ANY	64	X'00000040'
MQQF_CLWL_USEQ_LOCAL	128	X'00000080'

MQQMDT_* (Command format Queue Manager Definition Types)

Table 291. Values of constants		
Name	Decimal value	Hexadecimal value
MQQMDT_EXPLICIT_CLUSTER_SENDER	1	X'00000001'
MQQMDT_AUTO_CLUSTER_SENDER	2	X'00000002'
MQQMDT_AUTO_EXP_CLUSTER_SENDER	4	X'00000004'
MQQMDT_CLUSTER_RECEIVER	3	X'00000003'

MQQMF_* (Queue Manager Flags)

Table 292. Values of constants		
Name	Decimal value	Hexadecimal value
MQQMF_REPOSITORY_Q_MGR	2	X'00000002'
MQQMF_CLUSSDR_USER_DEFINED	8	X'00000008'
MQQMF_CLUSSDR_AUTO_DEFINED	16	X'00000010'
MQQMF_AVAILABLE	32	X'00000020'

MQQMFC_* (Command format Queue Manager Facility)

Table 293. Values of constants		
Name	Decimal value	Hexadecimal value
MQQMFC_IMS_BRIDGE	1	X'00000001'
MQQMFC_DB2	2	X'00000002'

MQQMSTA_* (Command format Queue Manager Status)

Table 294. Values of constants		
Name	Decimal value	Hexadecimal value
MQQMSTA_STARTING	1	X'00000001'
MQQMSTA_RUNNING	2	X'00000002'
MQQMSTA QUIESCING	3	X'00000003'

MQQMT_* (Command format Queue Manager Types)

Table 295. Values of constants		
Name	Decimal value	Hexadecimal value
MQQMT_NORMAL	0	X'00000000'
MQQMT_REPOSITORY	1	X'00000001'

MQQO_* (Command format Quiesce Options)

Table 296. Values of constants		
Name	Decimal value	Hexadecimal value
MQQO_YES	1	X'00000001'
MQQO_NO	0	X'00000000'

MQQSGD_* (Queue Sharing Group Dispositions)

Table 297. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSGD_ALL	-1	X'FFFFFFFF'
MQQSGD_Q_MGR	0	X'00000000'
MQQSGD_COPY	1	X'00000001'
MQQSGD_SHARED	2	X'00000002'
MQQSGD_GROUP	3	X'00000003'

Table 297. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQQSGD_PRIVATE	4	X'00000004'
MQQSGD_LIVE	6	X'00000006'

MQQSGS_* (Command format QSG Status)

Table 298. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSGS_UNKNOWN	0	X'00000000'
MQQSGS_CREATED	1	X'00000001'
MQQSGS_ACTIVE	2	X'00000002'
MQQSGS_INACTIVE	3	X'00000003'
MQQSGS_FAILED	4	X'00000004'
MQQSGS_PENDING	5	X'00000005'

MQQSIE_* (Command format Queue Service-Interval Events)

Table 299. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSIE_NONE	0	X'00000000'
MQQSIE_HIGH	1	X'00000001'
MQQSIE_OK	2	X'00000002'

MQQSO_* (Command format Queue Status Open Options for SET, BROWSE, INPUT)

Table 300. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSO_NO	0	X'00000000'
MQQSO_YES	1	X'00000001'
MQQSO_SHARED	1	X'00000001'
MQQSO_EXCLUSIVE	2	X'00000002'

MQQSOT_* (Command format Queue Status Open Types)

Table 301. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSOT_ALL	1	X'00000001'
MQQSOT_INPUT	2	X'00000002'
MQQSOT_OUTPUT	3	X'00000003'

MQQSUM_* (Command format Queue Status Uncommitted Messages)

Table 302. Values of constants		
Name	Decimal value	Hexadecimal value
MQQSUM_YES	1	X'00000001'

Table 302. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQQSUM_NO	0	X'00000000'

MQQT_* (Queue Types and Extended Queue Types)

Queue Types

Table 303. Values of constants		
Name	Decimal value	Hexadecimal value
MQQT_LOCAL	1	X'00000001'
MQQT_MODEL	2	X'00000002'
MQQT_ALIAS	3	X'00000003'
MQQT_REMOTE	6	X'00000006'
MQQT_CLUSTER	7	X'00000007'

Extended Queue Types

Table 304. Values of constants		
Name	Decimal value	Hexadecimal value
MQQT_ALL	1001	X'000003E9'

MQRC_* (Reason Codes)

Table 305. Values of constants		
Name	Decimal value	Hexadecimal value
MQRC_NONE	0	X'00000000'
MQRC_APPL_FIRST	900	X'00000384'
MQRC_APPL_LAST	999	X'000003E7'
MQRC_ALIAS_BASE_Q_TYPE_ERROR	2001	X'000007D1'
MQRC_ALREADY_CONNECTED	2002	X'000007D2'
MQRC_BACKED_OUT	2003	X'000007D3'
MQRC_BUFFER_ERROR	2004	X'000007D4'
MQRC_BUFFER_LENGTH_ERROR	2005	X'000007D5'
MQRC_CHAR_ATTR_LENGTH_ERROR	2006	X'000007D6'
MQRC_CHAR_ATTRS_ERROR	2007	X'000007D7'
MQRC_CHAR_ATTRS_TOO_SHORT	2008	X'000007D8'
MQRC_CONNECTION_BROKEN	2009	X'000007D9'
MQRC_DATA_LENGTH_ERROR	2010	X'000007DA'
MQRC_DYNAMIC_Q_NAME_ERROR	2011	X'000007DB'
MQRC_ENVIRONMENT_ERROR	2012	X'000007DC'
MQRC_EXPIRY_ERROR	2013	X'000007DD'
MQRC_FEEDBACK_ERROR	2014	X'000007DE'
MQRC_GET_INHIBITED	2016	X'000007E0'
MQRC_HANDLE_NOT_AVAILABLE	2017	X'000007E1'

Table 305. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQRC_HCONN_ERROR	2018	X'000007E2'
MQRC_HOBJ_ERROR	2019	X'000007E3'
MQRC_INHIBIT_VALUE_ERROR	2020	X'000007E4'
MQRC_INT_ATTR_COUNT_ERROR	2021	X'000007E5'
MQRC_INT_ATTR_COUNT_TOO_SMALL	2022	X'000007E6'
MQRC_INT_ATTRS_ARRAY_ERROR	2023	X'000007E7'
MQRC_SYNCPOINT_LIMIT_REACHED	2024	X'000007E8'
MQRC_MAX_CONNS_LIMIT_REACHED	2025	X'000007E9'
MQRC_MD_ERROR	2026	X'000007EA'
MQRC_MISSING_REPLY_TO_Q	2027	X'000007EB'
MQRC_MSG_TYPE_ERROR	2029	X'000007ED'
MQRC_MSG_TOO_BIG_FOR_Q	2030	X'000007EE'
MQRC_MSG_TOO_BIG_FOR_Q_MGR	2031	X'000007EF'
MQRC_NO_MSG_AVAILABLE	2033	X'000007F1'
MQRC_NO_MSG_UNDER_CURSOR	2034	X'000007F2'
MQRC_NOT_AUTHORIZED	2035	X'000007F3'
MQRC_NOT_OPEN_FOR_BROWSE	2036	X'000007F4'
MQRC_NOT_OPEN_FOR_INPUT	2037	X'000007F5'
MQRC_NOT_OPEN_FOR_INQUIRE	2038	X'000007F6'
MQRC_NOT_OPEN_FOR_OUTPUT	2039	X'000007F7'
MQRC_NOT_OPEN_FOR_SET	2040	X'000007F8'
MQRC_OBJECT_CHANGED	2041	X'000007F9'
MQRC_OBJECT_IN_USE	2042	X'000007FA'
MQRC_OBJECT_TYPE_ERROR	2043	X'000007FB'
MQRC_OD_ERROR	2044	X'000007FC'
MQRC_OPTION_NOT_VALID_FOR_TYPE	2045	X'000007FD'
MQRC_OPTIONS_ERROR	2046	X'000007FE'
MQRC_PERSISTENCE_ERROR	2047	X'000007FF'
MQRC_PERSISTENT_NOT_ALLOWED	2048	X'00000800'
MQRC_PRIORITY_EXCEEDS_MAXIMUM	2049	X'00000801'
MQRC_PRIORITY_ERROR	2050	X'00000802'
MQRC_PUT_INHIBITED	2051	X'00000803'
MQRC_Q_DELETED	2052	X'00000804'
MQRC_Q_FULL	2053	X'00000805'
MQRC_Q_NOT_EMPTY	2055	X'00000807'
MQRC_Q_SPACE_NOT_AVAILABLE	2056	X'00000808'
MQRC_Q_TYPE_ERROR	2057	X'00000809'
MQRC_Q_MGR_NAME_ERROR	2058	X'0000080A'
MQRC_Q_MGR_NOT_AVAILABLE	2059	X'0000080B'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_REPORT_OPTIONS_ERROR	2061	X'0000080D'
MQRC_SECOND_MARK_NOT_ALLOWED	2062	X'0000080E'
MQRC_SECURITY_ERROR	2063	X'0000080F'
MQRC_SELECTOR_COUNT_ERROR	2065	X'00000811'
MQRC_SELECTOR_LIMIT_EXCEEDED	2066	X'00000812'
MQRC_SELECTOR_ERROR	2067	X'00000813'
MQRC_SELECTOR_NOT_FOR_TYPE	2068	X'00000814'
MQRC_SIGNAL_OUTSTANDING	2069	X'00000815'
MQRC_SIGNAL_REQUEST_ACCEPTED	2070	X'00000816'
MQRC_STORAGE_NOT_AVAILABLE	2071	X'00000817'
MQRC_SYNCPOINT_NOT_AVAILABLE	2072	X'00000818'
MQRC_TRIGGER_CONTROL_ERROR	2075	X'0000081B'
MQRC_TRIGGER_DEPTH_ERROR	2076	X'0000081C'
MQRC_TRIGGER_MSG_PRIORITY_ERR	2077	X'0000081D'
MQRC_TRIGGER_TYPE_ERROR	2078	X'0000081E'
MQRC_TRUNCATED_MSG_ACCEPTED	2079	X'0000081F'
MQRC_TRUNCATED_MSG_FAILED	2080	X'00000820'
MQRC_UNKNOWN_ALIAS_BASE_Q	2082	X'00000822'
MQRC_UNKNOWN_OBJECT_NAME	2085	X'00000825'
MQRC_UNKNOWN_OBJECT_Q_MGR	2086	X'00000826'
MQRC_UNKNOWN_REMOTE_Q_MGR	2087	X'00000827'
MQRC_WAIT_INTERVAL_ERROR	2090	X'0000082A'
MQRC_XMIT_Q_TYPE_ERROR	2091	X'0000082B'
MQRC_XMIT_Q_USAGE_ERROR	2092	X'0000082C'
MQRC_NOT_OPEN_FOR_PASS_ALL	2093	X'0000082D'
MQRC_NOT_OPEN_FOR_PASS_IDENT	2094	X'0000082E'
MQRC_NOT_OPEN_FOR_SET_ALL	2095	X'0000082F'
MQRC_NOT_OPEN_FOR_SET_IDENT	2096	X'00000830'
MQRC_CONTEXT_HANDLE_ERROR	2097	X'00000831'
MQRC_CONTEXT_NOT_AVAILABLE	2098	X'00000832'
MQRC_SIGNAL1_ERROR	2099	X'00000833'
MQRC_OBJECT_ALREADY_EXISTS	2100	X'00000834'
MQRC_OBJECT_DAMAGED	2101	X'00000835'
MQRC_RESOURCE_PROBLEM	2102	X'00000836'
MQRC_ANOTHER_Q_MGR_CONNECTED	2103	X'00000837'
MQRC_UNKNOWN_REPORT_OPTION	2104	X'00000838'
MQRC_STORAGE_CLASS_ERROR	2105	X'00000839'
MQRC_COD_NOT_VALID_FOR_XCF_Q	2106	X'0000083A'
MQRC_XWAIT_CANCELED	2107	X'0000083B'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_XWAIT_ERROR	2108	X'0000083C'
MQRC_SUPPRESSED_BY_EXIT	2109	X'0000083D'
MQRC_FORMAT_ERROR	2110	X'0000083E'
MQRC_SOURCE_CCSID_ERROR	2111	X'0000083F'
MQRC_SOURCE_INTEGER_ENC_ERROR	2112	X'00000840'
MQRC_SOURCE_DECIMAL_ENC_ERROR	2113	X'00000841'
MQRC_SOURCE_FLOAT_ENC_ERROR	2114	X'00000842'
MQRC_TARGET_CCSID_ERROR	2115	X'00000843'
MQRC_TARGET_INTEGER_ENC_ERROR	2116	X'00000844'
MQRC_TARGET_DECIMAL_ENC_ERROR	2117	X'00000845'
MQRC_TARGET_FLOAT_ENC_ERROR	2118	X'00000846'
MQRC_NOT_CONVERTED	2119	X'00000847'
MQRC_CONVERTED_MSG_TOO_BIG	2120	X'00000848'
MQRC_TRUNCATED	2120	X'00000848'
MQRC_NO_EXTERNAL_PARTICIPANTS	2121	X'00000849'
MQRC_PARTICIPANT_NOT_AVAILABLE	2122	X'0000084A'
MQRC_OUTCOME_MIXED	2123	X'0000084B'
MQRC_OUTCOME_PENDING	2124	X'0000084C'
MQRC_BRIDGE_STARTED	2125	X'0000084D'
MQRC_BRIDGE_STOPPED	2126	X'0000084E'
MQRC_ADAPTER_STORAGE_SHORTAGE	2127	X'0000084F'
MQRC_UOW_IN_PROGRESS	2128	X'00000850'
MQRC_ADAPTER_CONN_LOAD_ERROR	2129	X'00000851'
MQRC_ADAPTER_SERV_LOAD_ERROR	2130	X'00000852'
MQRC_ADAPTER_DEFS_ERROR	2131	X'00000853'
MQRC_ADAPTER_DEFS_LOAD_ERROR	2132	X'00000854'
MQRC_ADAPTER_CONV_LOAD_ERROR	2133	X'00000855'
MQRC_BO_ERROR	2134	X'00000856'
MQRC_DH_ERROR	2135	X'00000857'
MQRC_MULTIPLE_REASONS	2136	X'00000858'
MQRC_OPEN_FAILED	2137	X'00000859'
MQRC_ADAPTER_DISC_LOAD_ERROR	2138	X'0000085A'
MQRC_CNO_ERROR	2139	X'0000085B'
MQRC_CICS_WAIT_FAILED	2140	X'0000085C'
MQRC_DLH_ERROR	2141	X'0000085D'
MQRC_HEADER_ERROR	2142	X'0000085E'
MQRC_SOURCE_LENGTH_ERROR	2143	X'0000085F'
MQRC_TARGET_LENGTH_ERROR	2144	X'00000860'
MQRC_SOURCE_BUFFER_ERROR	2145	X'00000861'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_TARGET_BUFFER_ERROR	2146	X'00000862'
MQRC_IIH_ERROR	2148	X'00000864'
MQRC_PCF_ERROR	2149	X'00000865'
MQRC_DBCS_ERROR	2150	X'00000866'
MQRC_OBJECT_NAME_ERROR	2152	X'00000868'
MQRC_OBJECT_Q_MGR_NAME_ERROR	2153	X'00000869'
MQRC_RECS_PRESENT_ERROR	2154	X'0000086A'
MQRC_OBJECT_RECORDS_ERROR	2155	X'0000086B'
MQRC_RESPONSE_RECORDS_ERROR	2156	X'0000086C'
MQRC_ASID_MISMATCH	2157	X'0000086D'
MQRC_PMO_RECORD_FLAGS_ERROR	2158	X'0000086E'
MQRC_PUT_MSG_RECORDS_ERROR	2159	X'0000086F'
MQRC_CONN_ID_IN_USE	2160	X'00000870'
MQRC_Q_MGR QUIESCING	2161	X'00000871'
MQRC_Q_MGR_STOPPING	2162	X'00000872'
MQRC_DUPLICATE_RECOV_COORD	2163	X'00000873'
MQRC_PMO_ERROR	2173	X'0000087D'
MQRC_API_EXIT_NOT_FOUND	2182	X'00000886'
MQRC_API_EXIT_LOAD_ERROR	2183	X'00000887'
MQRC_REMOTE_Q_NAME_ERROR	2184	X'00000888'
MQRC_INCONSISTENT_PERSISTENCE	2185	X'00000889'
MQRC_GMO_ERROR	2186	X'0000088A'
MQRC_CICS_BRIDGE_RESTRICTION	2187	X'0000088B'
MQRC_STOPPED_BY_CLUSTER_EXIT	2188	X'0000088C'
MQRC_CLUSTER_RESOLUTION_ERROR	2189	X'0000088D'
MQRC_CONVERTED_STRING_TOO_BIG	2190	X'0000088E'
MQRC_TMC_ERROR	2191	X'0000088F'
MQRC_PAGESET_FULL	2192	X'00000890'
MQRC_STORAGE_MEDIUM_FULL	2192	X'00000890'
MQRC_PAGESET_ERROR	2193	X'00000891'
MQRC_NAME_NOT_VALID_FOR_TYPE	2194	X'00000892'
MQRC_UNEXPECTED_ERROR	2195	X'00000893'
MQRC_UNKNOWN_XMIT_Q	2196	X'00000894'
MQRC_UNKNOWN_DEF_XMIT_Q	2197	X'00000895'
MQRC_DEF_XMIT_Q_TYPE_ERROR	2198	X'00000896'
MQRC_DEF_XMIT_Q_USAGE_ERROR	2199	X'00000897'
MQRC_MSG_MARKED_BROWSE_CO_OP	2200	X'00000898'
MQRC_NAME_IN_USE	2201	X'00000899'
MQRC_CONNECTION QUIESCING	2202	X'0000089A'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_CONNECTION_STOPPING	2203	X'0000089B'
MQRC_ADAPTER_NOT_AVAILABLE	2204	X'0000089C'
MQRC_MSG_ID_ERROR	2206	X'0000089E'
MQRC_CORREL_ID_ERROR	2207	X'0000089F'
MQRC_FILE_SYSTEM_ERROR	2208	X'000008A0'
MQRC_NO_MSG_LOCKED	2209	X'000008A1'
MQRC_SOAP_DOTNET_ERROR	2210	X'000008A2'
MQRC_SOAP_AXIS_ERROR	2211	X'000008A3'
MQRC_SOAP_URL_ERROR	2212	X'000008A4'
MQRC_FILE_NOT_AUDITED	2216	X'000008A8'
MQRC_CONNECTION_NOT_AUTHORIZED	2217	X'000008A9'
MQRC_MSG_TOO_BIG_FOR_CHANNEL	2218	X'000008AA'
MQRC_CALL_IN_PROGRESS	2219	X'000008AB'
MQRC_RMH_ERROR	2220	X'000008AC'
MQRC_Q_MGR_ACTIVE	2222	X'000008AE'
MQRC_Q_MGR_NOT_ACTIVE	2223	X'000008AF'
MQRC_Q_DEPTH_HIGH	2224	X'000008B0'
MQRC_Q_DEPTH_LOW	2225	X'000008B1'
MQRC_Q_SERVICE_INTERVAL_HIGH	2226	X'000008B2'
MQRC_Q_SERVICE_INTERVAL_OK	2227	X'000008B3'
MQRC_RFH_HEADER_FIELD_ERROR	2228	X'000008B4'
MQRC_RAS_PROPERTY_ERROR	2229	X'000008B5'
MQRC_UNIT_OF_WORK_NOT_STARTED	2232	X'000008B8'
MQRC_CHANNEL_AUTO_DEF_OK	2233	X'000008B9'
MQRC_CHANNEL_AUTO_DEF_ERROR	2234	X'000008BA'
MQRC_CFH_ERROR	2235	X'000008BB'
MQRC_CFIL_ERROR	2236	X'000008BC'
MQRC_CFIN_ERROR	2237	X'000008BD'
MQRC_CFSL_ERROR	2238	X'000008BE'
MQRC_CFST_ERROR	2239	X'000008BF'
MQRC_INCOMPLETE_GROUP	2241	X'000008C1'
MQRC_INCOMPLETE_MSG	2242	X'000008C2'
MQRC_INCONSISTENT_CCSDS	2243	X'000008C3'
MQRC_INCONSISTENT_ENCODINGS	2244	X'000008C4'
MQRC_INCONSISTENT_UOW	2245	X'000008C5'
MQRC_INVALID_MSG_UNDER_CURSOR	2246	X'000008C6'
MQRC_MATCH_OPTIONS_ERROR	2247	X'000008C7'
MQRC_MDE_ERROR	2248	X'000008C8'
MQRC_MSG_FLAGS_ERROR	2249	X'000008C9'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_MSG_SEQ_NUMBER_ERROR	2250	X'000008CA'
MQRC_OFFSET_ERROR	2251	X'000008CB'
MQRC_ORIGINAL_LENGTH_ERROR	2252	X'000008CC'
MQRC_SEGMENT_LENGTH_ZERO	2253	X'000008CD'
MQRC_UOW_NOT_AVAILABLE	2255	X'000008CF'
MQRC_WRONG_GMO_VERSION	2256	X'000008D0'
MQRC_WRONG_MD_VERSION	2257	X'000008D1'
MQRC_GROUP_ID_ERROR	2258	X'000008D2'
MQRC_INCONSISTENT_BROWSE	2259	X'000008D3'
MQRC_XQH_ERROR	2260	X'000008D4'
MQRC_SRC_ENV_ERROR	2261	X'000008D5'
MQRC_SRC_NAME_ERROR	2262	X'000008D6'
MQRC_DEST_ENV_ERROR	2263	X'000008D7'
MQRC_DEST_NAME_ERROR	2264	X'000008D8'
MQRC_TM_ERROR	2265	X'000008D9'
MQRC_CLUSTER_EXIT_ERROR	2266	X'000008DA'
MQRC_CLUSTER_EXIT_LOAD_ERROR	2267	X'000008DB'
MQRC_CLUSTER_PUT_INHIBITED	2268	X'000008DC'
MQRC_CLUSTER_RESOURCE_ERROR	2269	X'000008DD'
MQRC_NO_DESTINATIONS_AVAILABLE	2270	X'000008DE'
MQRC_CONN_TAG_IN_USE	2271	X'000008DF'
MQRC_PARTIALLY_CONVERTED	2272	X'000008E0'
MQRC_CONNECTION_ERROR	2273	X'000008E1'
MQRC_OPTION_ENVIRONMENT_ERROR	2274	X'000008E2'
MQRC_CD_ERROR	2277	X'000008E5'
MQRC_CLIENT_CONN_ERROR	2278	X'000008E6'
MQRC_CHANNEL_STOPPED_BY_USER	2279	X'000008E7'
MQRC_HCONFIG_ERROR	2280	X'000008E8'
MQRC_FUNCTION_ERROR	2281	X'000008E9'
MQRC_CHANNEL_STARTED	2282	X'000008EA'
MQRC_CHANNEL_STOPPED	2283	X'000008EB'
MQRC_CHANNEL_CONV_ERROR	2284	X'000008EC'
MQRC_SERVICE_NOT_AVAILABLE	2285	X'000008ED'
MQRC_INITIALIZATION_FAILED	2286	X'000008EE'
MQRC_TERMINATION_FAILED	2287	X'000008EF'
MQRC_UNKNOWN_Q_NAME	2288	X'000008F0'
MQRC_SERVICE_ERROR	2289	X'000008F1'
MQRC_Q_ALREADY_EXISTS	2290	X'000008F2'
MQRC_USER_ID_NOT_AVAILABLE	2291	X'000008F3'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_UNKNOWN_ENTITY	2292	X'000008F4'
MQRC_UNKNOWN_AUTH_ENTITY	2293	X'000008F5'
MQRC_UNKNOWN_REF_OBJECT	2294	X'000008F6'
MQRC_CHANNEL_ACTIVATED	2295	X'000008F7'
MQRC_CHANNEL_NOT_ACTIVATED	2296	X'000008F8'
MQRC_UOW_CANCELED	2297	X'000008F9'
MQRC_FUNCTION_NOT_SUPPORTED	2298	X'000008FA'
MQRC_SELECTOR_TYPE_ERROR	2299	X'000008FB'
MQRC_COMMAND_TYPE_ERROR	2300	X'000008FC'
MQRC_MULTIPLE_INSTANCE_ERROR	2301	X'000008FD'
MQRC_SYSTEM_ITEM_NOT_ALTERABLE	2302	X'000008FE'
MQRC_BAG_CONVERSION_ERROR	2303	X'000008FF'
MQRC_SELECTOR_OUT_OF_RANGE	2304	X'00000900'
MQRC_SELECTOR_NOT_UNIQUE	2305	X'00000901'
MQRC_INDEX_NOT_PRESENT	2306	X'00000902'
MQRC_STRING_ERROR	2307	X'00000903'
MQRC_ENCODING_NOT_SUPPORTED	2308	X'00000904'
MQRC_SELECTOR_NOT_PRESENT	2309	X'00000905'
MQRC_OUT_SELECTOR_ERROR	2310	X'00000906'
MQRC_STRING_TRUNCATED	2311	X'00000907'
MQRC_SELECTOR_WRONG_TYPE	2312	X'00000908'
MQRC_INCONSISTENT_ITEM_TYPE	2313	X'00000909'
MQRC_INDEX_ERROR	2314	X'0000090A'
MQRC_SYSTEM_BAG_NOT_ALTERABLE	2315	X'0000090B'
MQRC_ITEM_COUNT_ERROR	2316	X'0000090C'
MQRC_FORMAT_NOT_SUPPORTED	2317	X'0000090D'
MQRC_SELECTOR_NOT_SUPPORTED	2318	X'0000090E'
MQRC_ITEM_VALUE_ERROR	2319	X'0000090F'
MQRC_HBAG_ERROR	2320	X'00000910'
MQRC_PARAMETER_MISSING	2321	X'00000911'
MQRC_CMD_SERVER_NOT_AVAILABLE	2322	X'00000912'
MQRC_STRING_LENGTH_ERROR	2323	X'00000913'
MQRC_INQUIRY_COMMAND_ERROR	2324	X'00000914'
MQRC_NESTED_BAG_NOT_SUPPORTED	2325	X'00000915'
MQRC_BAG_WRONG_TYPE	2326	X'00000916'
MQRC_ITEM_TYPE_ERROR	2327	X'00000917'
MQRC_SYSTEM_BAG_NOT_DELETABLE	2328	X'00000918'
MQRC_SYSTEM_ITEM_NOT_DELETABLE	2329	X'00000919'
MQRC_CODED_CHAR_SET_ID_ERROR	2330	X'0000091A'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_MSG_TOKEN_ERROR	2331	X'0000091B'
MQRC_MISSING_WIH	2332	X'0000091C'
MQRC_WIH_ERROR	2333	X'0000091D'
MQRC_RFH_ERROR	2334	X'0000091E'
MQRC_RFH_STRING_ERROR	2335	X'0000091F'
MQRC_RFH_COMMAND_ERROR	2336	X'00000920'
MQRC_RFH_PARM_ERROR	2337	X'00000921'
MQRC_RFH_DUPLICATE_PARM	2338	X'00000922'
MQRC_RFH_PARM_MISSING	2339	X'00000923'
MQRC_CHAR_CONVERSION_ERROR	2340	X'00000924'
MQRC_UCS2_CONVERSION_ERROR	2341	X'00000925'
MQRC_DB2_NOT_AVAILABLE	2342	X'00000926'
MQRC_OBJECT_NOT_UNIQUE	2343	X'00000927'
MQRC_CONN_TAG_NOT_RELEASED	2344	X'00000928'
MQRC_CF_NOT_AVAILABLE	2345	X'00000929'
MQRC_CF_STRUC_IN_USE	2346	X'0000092A'
MQRC_CF_STRUC_LIST_HDR_IN_USE	2347	X'0000092B'
MQRC_CF_STRUC_AUTH_FAILED	2348	X'0000092C'
MQRC_CF_STRUC_ERROR	2349	X'0000092D'
MQRC_CONN_TAG_NOT_USABLE	2350	X'0000092E'
MQRC_GLOBAL_UOW_CONFLICT	2351	X'0000092F'
MQRC_LOCAL_UOW_CONFLICT	2352	X'00000930'
MQRC_HANDLE_IN_USE_FOR_UOW	2353	X'00000931'
MQRC_UOW_ENLISTMENT_ERROR	2354	X'00000932'
MQRC_UOW_MIX_NOT_SUPPORTED	2355	X'00000933'
MQRC_WXP_ERROR	2356	X'00000934'
MQRC_CURRENT_RECORD_ERROR	2357	X'00000935'
MQRC_NEXT_OFFSET_ERROR	2358	X'00000936'
MQRC_NO_RECORD_AVAILABLE	2359	X'00000937'
MQRC_OBJECT_LEVEL_INCOMPATIBLE	2360	X'00000938'
MQRC_NEXT_RECORD_ERROR	2361	X'00000939'
MQRC_BACKOUT_THRESHOLD_REACHED	2362	X'0000093A'
MQRC_MSG_NOT_MATCHED	2363	X'0000093B'
MQRC_JMS_FORMAT_ERROR	2364	X'0000093C'
MQRC_SEGMENTS_NOT_SUPPORTED	2365	X'0000093D'
MQRC_WRONG_CF_LEVEL	2366	X'0000093E'
MQRC_CONFIG_CREATE_OBJECT	2367	X'0000093F'
MQRC_CONFIG_CHANGE_OBJECT	2368	X'00000940'
MQRC_CONFIG_DELETE_OBJECT	2369	X'00000941'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_CONFIG_REFRESH_OBJECT	2370	X'00000942'
MQRC_CHANNEL_SSL_ERROR	2371	X'00000943'
MQRC_PARTICIPANT_NOT_DEFINED	2372	X'00000944'
MQRC_CF_STRUC_FAILED	2373	X'00000945'
MQRC_API_EXIT_ERROR	2374	X'00000946'
MQRC_API_EXIT_INIT_ERROR	2375	X'00000947'
MQRC_API_EXIT_TERM_ERROR	2376	X'00000948'
MQRC_EXIT_REASON_ERROR	2377	X'00000949'
MQRC_RESERVED_VALUE_ERROR	2378	X'0000094A'
MQRC_NO_DATA_AVAILABLE	2379	X'0000094B'
MQRC_SCO_ERROR	2380	X'0000094C'
MQRC_KEY_REPOSITORY_ERROR	2381	X'0000094D'
MQRC_CRYPTO_HARDWARE_ERROR	2382	X'0000094E'
MQRC_AUTH_INFO_REC_COUNT_ERROR	2383	X'0000094F'
MQRC_AUTH_INFO_REC_ERROR	2384	X'00000950'
MQRC_AIR_ERROR	2385	X'00000951'
MQRC_AUTH_INFO_TYPE_ERROR	2386	X'00000952'
MQRC_AUTH_INFO_CONN_NAME_ERROR	2387	X'00000953'
MQRC_LDAP_USER_NAME_ERROR	2388	X'00000954'
MQRC_LDAP_USER_NAME_LENGTH_ERR	2389	X'00000955'
MQRC_LDAP_PASSWORD_ERROR	2390	X'00000956'
MQRC_SSL_ALREADY_INITIALIZED	2391	X'00000957'
MQRC_SSL_CONFIG_ERROR	2392	X'00000958'
MQRC_SSL_INITIALIZATION_ERROR	2393	X'00000959'
MQRC_Q_INDEX_TYPE_ERROR	2394	X'0000095A'
MQRC_CFBS_ERROR	2395	X'0000095B'
MQRC_SSL_NOT_ALLOWED	2396	X'0000095C'
MQRC_JSSE_ERROR	2397	X'0000095D'
MQRC_SSL_PEER_NAME_MISMATCH	2398	X'0000095E'
MQRC_SSL_PEER_NAME_ERROR	2399	X'0000095F'
MQRC_UNSUPPORTED_CIPHER_SUITE	2400	X'00000960'
MQRC_SSL_CERTIFICATE_REVOKED	2401	X'00000961'
MQRC_SSL_CERT_STORE_ERROR	2402	X'00000962'
MQRC_CLIENT_EXIT_LOAD_ERROR	2406	X'00000966'
MQRC_CLIENT_EXIT_ERROR	2407	X'00000967'
MQRC_UOW_COMMITTED	2408	X'00000968'
MQRC_SSL_KEY_RESET_ERROR	2409	X'00000969'
MQRC_UNKNOWN_COMPONENT_NAME	2410	X'0000096A'
MQRC_LOGGER_STATUS	2411	X'0000096B'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_COMMAND_MQSC	2412	X'0000096C'
MQRC_COMMAND_PCF	2413	X'0000096D'
MQRC_CFIF_ERROR	2414	X'0000096E'
MQRC_CFSF_ERROR	2415	X'0000096F'
MQRC_CFGR_ERROR	2416	X'00000970'
MQRC_MSG_NOT_ALLOWED_IN_GROUP	2417	X'00000971'
MQRC_FILTER_OPERATOR_ERROR	2418	X'00000972'
MQRC_NESTED_SELECTOR_ERROR	2419	X'00000973'
MQRC_EPH_ERROR	2420	X'00000974'
MQRC_RFH_FORMAT_ERROR	2421	X'00000975'
MQRC_CFBF_ERROR	2422	X'00000976'
MQRC_CLIENT_CHANNEL_CONFLICT	2423	X'00000977'
MQRC_SD_ERROR	2424	X'00000978'
MQRC_TOPIC_STRING_ERROR	2425	X'00000979'
MQRC_STS_ERROR	2426	X'0000097A'
MQRC_NO_SUBSCRIPTION	2428	X'0000097C'
MQRC_SUBSCRIPTION_IN_USE	2429	X'0000097D'
MQRC_STAT_TYPE_ERROR	2430	X'0000097E'
MQRC_SUB_USER_DATA_ERROR	2431	X'0000097F'
MQRC_SUB_ALREADY_EXISTS	2432	X'00000980'
MQRC_IDENTITY_MISMATCH	2434	X'00000982'
MQRC_ALTER_SUB_ERROR	2435	X'00000983'
MQRC_DURABILITY_NOT_ALLOWED	2436	X'00000984'
MQRC_NO_RETAINED_MSG	2437	X'00000985'
MQRC_SRO_ERROR	2438	X'00000986'
MQRC_SUB_NAME_ERROR	2440	X'00000988'
MQRC_OBJECT_STRING_ERROR	2441	X'00000989'
MQRC_PROPERTY_NAME_ERROR	2442	X'0000098A'
MQRC_SEGMENTATION_NOT_ALLOWED	2443	X'0000098B'
MQRC_CBD_ERROR	2444	X'0000098C'
MQRC_CTLO_ERROR	2445	X'0000098D'
MQRC_NO_CALLBACKS_ACTIVE	2446	X'0000098E'
MQRC_CALLBACK_NOT_REGISTERED	2448	X'00000990'
MQRC_OPTIONS_CHANGED	2457	X'00000999'
MQRC_READ_AHEAD_MSGS	2458	X'0000099A'
MQRC_SELECTOR_SYNTAX_ERROR	2459	X'0000099B'
MQRC_HMSG_ERROR	2460	X'0000099C'
MQRC_CMHO_ERROR	2461	X'0000099D'
MQRC_DMHO_ERROR	2462	X'0000099E'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_SMPO_ERROR	2463	X'0000099F'
MQRC_IMPO_ERROR	2464	X'000009A0'
MQRC_PROPERTY_NAME_TOO_BIG	2465	X'000009A1'
MQRC_PROP_VALUE_NOT_CONVERTED	2466	X'000009A2'
MQRC_PROP_TYPE_NOT_SUPPORTED	2467	X'000009A3'
MQRC_PROPERTY_VALUE_TOO_BIG	2469	X'000009A5'
MQRC_PROP_CONV_NOT_SUPPORTED	2470	X'000009A6'
MQRC_PROPERTY_NOT_AVAILABLE	2471	X'000009A7'
MQRC_PROP_NUMBER_FORMAT_ERROR	2472	X'000009A8'
MQRC_PROPERTY_TYPE_ERROR	2473	X'000009A9'
MQRC_PROPERTIES_TOO_BIG	2478	X'000009AE'
MQRC_PUT_NOT_RETAINED	2479	X'000009AF'
MQRC_ALIAS_TARGTYPE_CHANGED	2480	X'000009B0'
MQRC_DMPO_ERROR	2481	X'000009B1'
MQRC_PD_ERROR	2482	X'000009B2'
MQRC_CALLBACK_TYPE_ERROR	2483	X'000009B3'
MQRC_CBD_OPTIONS_ERROR	2484	X'000009B4'
MQRC_MAX_MSG_LENGTH_ERROR	2485	X'000009B5'
MQRC_CALLBACK_ROUTINE_ERROR	2486	X'000009B6'
MQRC_CALLBACK_LINK_ERROR	2487	X'000009B7'
MQRC_OPERATION_ERROR	2488	X'000009B8'
MQRC_BMHO_ERROR	2489	X'000009B9'
MQRC_UNSUPPORTED_PROPERTY	2490	X'000009BA'
MQRC_PROP_NAME_NOT_CONVERTED	2492	X'000009BC'
MQRC_GET_ENABLED	2494	X'000009BE'
MQRC_MODULE_NOT_FOUND	2495	X'000009BF'
MQRC_MODULE_INVALID	2496	X'000009C0'
MQRC_MODULE_ENTRY_NOT_FOUND	2497	X'000009C1'
MQRC_MIXED_CONTENT_NOT_ALLOWED	2498	X'000009C2'
MQRC_MSG_HANDLE_IN_USE	2499	X'000009C3'
MQRC_HCONN_ASYNC_ACTIVE	2500	X'000009C4'
MQRC_MHBO_ERROR	2501	X'000009C5'
MQRC_PUBLICATION_FAILURE	2502	X'000009C6'
MQRC_SUB_INHIBITED	2503	X'000009C7'
MQRC_SELECTOR_ALWAYS_FALSE	2504	X'000009C8'
MQRC_XEPO_ERROR	2507	X'000009CB'
MQRC_DURABILITY_NOT_ALTERABLE	2509	X'000009CD'
MQRC_TOPIC_NOT_ALTERABLE	2510	X'000009CE'
MQRC_SUBLEVEL_NOT_ALTERABLE	2512	X'000009D0'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_PROPERTY_NAME_LENGTH_ERR	2513	X'000009D1'
MQRC_DUPLICATE_GROUP_SUB	2514	X'000009D2'
MQRC_GROUPING_NOT_ALTERABLE	2515	X'000009D3'
MQRC_SELECTOR_INVALID_FOR_TYPE	2516	X'000009D4'
MQRC_HOBJ QUIESCED	2517	X'000009D5'
MQRC_HOBJ QUIESCED_NO_MSGS	2518	X'000009D6'
MQRC_SELECTION_STRING_ERROR	2519	X'000009D7'
MQRC_RES_OBJECT_STRING_ERROR	2520	X'000009D8'
MQRC_CONNECTION_SUSPENDED	2521	X'000009D9'
MQRC_INVALID_DESTINATION	2522	X'000009DA'
MQRC_INVALID_SUBSCRIPTION	2523	X'000009DB'
MQRC_SELECTOR_NOT_ALTERABLE	2524	X'000009DC'
MQRC_RETAINED_MSG_Q_ERROR	2525	X'000009DD'
MQRC_RETAINED_NOT_DELIVERED	2526	X'000009DE'
MQRC_RFH_RESTRICTED_FORMAT_ERR	2527	X'000009DF'
MQRC_CONNECTION_STOPPED	2528	X'000009E0'
MQRC_ASYNC_UOW_CONFLICT	2529	X'000009E1'
MQRC_ASYNC_XA_CONFLICT	2530	X'000009E2'
MQRC_PUBSUB_INHIBITED	2531	X'000009E3'
MQRC_MSG_HANDLE_COPY_FAILURE	2532	X'000009E4'
MQRC_DEST_CLASS_NOT_ALTERABLE	2533	X'000009E5'
MQRC_OPERATION_NOT_ALLOWED	2534	X'000009E6'
MQRC_ACTION_ERROR	2535	X'000009E7'
MQRC_CHANNEL_NOT_AVAILABLE	2537	X'000009E9'
MQRC_HOST_NOT_AVAILABLE	2538	X'000009EA'
MQRC_CHANNEL_CONFIG_ERROR	2539	X'000009EB'
MQRC_UNKNOWN_CHANNEL_NAME	2540	X'000009EC'
MQRC_LOOPING_PUBLICATION	2541	X'000009ED'
MQRC_ALREADY_JOINED	2542	X'000009EE'
MQRC_CHANNEL_SSL_WARNING	2552	X'000009F8'
MQRC_OCSP_URL_ERROR	2553	X'000009F9'
MQRC_CIPHER_SPEC_NOT_SUITE_B	2591	X'00000A1F'
MQRC_SUITE_B_ERROR	2592	X'00000A20'
MQRC_PASSWORD_PROTECTION_ERROR	2594	X'00000A22'
MQRC_REOPEN_EXCL_INPUT_ERROR	6100	X'000017D4'
MQRC_REOPEN_INQUIRE_ERROR	6101	X'000017D5'
MQRC_REOPEN_SAVED_CONTEXT_ERR	6102	X'000017D6'
MQRC_REOPEN_TEMPORARY_Q_ERROR	6103	X'000017D7'
MQRC_ATTRIBUTE_LOCKED	6104	X'000017D8'

Table 305. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRC_CURSOR_NOT_VALID	6105	X'000017D9'
MQRC_ENCODING_ERROR	6106	X'000017DA'
MQRC_STRUC_ID_ERROR	6107	X'000017DB'
MQRC_NULL_POINTER	6108	X'000017DC'
MQRC_NO_CONNECTION_REFERENCE	6109	X'000017DD'
MQRC_NO_BUFFER	6110	X'000017DE'
MQRC_BINARY_DATA_LENGTH_ERROR	6111	X'000017DF'
MQRC_BUFFER_NOT_AUTOMATIC	6112	X'000017E0'
MQRC_INSUFFICIENT_BUFFER	6113	X'000017E1'
MQRC_INSUFFICIENT_DATA	6114	X'000017E2'
MQRC_DATA_TRUNCATED	6115	X'000017E3'
MQRC_ZERO_LENGTH	6116	X'000017E4'
MQRC_NEGATIVE_LENGTH	6117	X'000017E5'
MQRC_NEGATIVE_OFFSET	6118	X'000017E6'
MQRC_INCONSISTENT_FORMAT	6119	X'000017E7'
MQRC_INCONSISTENT_OBJECT_STATE	6120	X'000017E8'
MQRC_CONTEXT_OBJECT_NOT_VALID	6121	X'000017E9'
MQRC_CONTEXT_OPEN_ERROR	6122	X'000017EA'
MQRC_STRUC_LENGTH_ERROR	6123	X'000017EB'
MQRC_NOT_CONNECTED	6124	X'000017EC'
MQRC_NOT_OPEN	6125	X'000017ED'
MQRC_DISTRIBUTION_LIST_EMPTY	6126	X'000017EE'
MQRC_INCONSISTENT_OPEN_OPTIONS	6127	X'000017EF'
MQRC_WRONG_VERSION	6128	X'000017F0'
MQRC_REFERENCE_ERROR	6129	X'000017F1'

MQRCCF_* (Command format header Reason Codes)

See [PCF reason codes](#) for more information on the programmer response.

Table 306. Values of constants		
Name	Decimal value	Hexadecimal value
MQRCCF_CFH_TYPE_ERROR	3001	X'00000BB9'
MQRCCF_CFH_LENGTH_ERROR	3002	X'00000BBA'
MQRCCF_CFH_VERSION_ERROR	3003	X'00000BBB'
MQRCCF_CFH_MSG_SEQ_NUMBER_ERR	3004	X'00000BBC'
MQRCCF_CFH_CONTROL_ERROR	3005	X'00000BBD'
MQRCCF_CFH_PARM_COUNT_ERROR	3006	X'00000BBE'
MQRCCF_CFH_COMMAND_ERROR	3007	X'00000BBF'
MQRCCF_COMMAND_FAILED	3008	X'00000BC0'
MQRCCF_CFIN_LENGTH_ERROR	3009	X'00000BC1'

Table 306. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRCCF_CFST_LENGTH_ERROR	3010	X'00000BC2'
MQRCCF_CFST_STRING_LENGTH_ERR	3011	X'00000BC3'
MQRCCF_FORCE_VALUE_ERROR	3012	X'00000BC4'
MQRCCF_STRUCTURE_TYPE_ERROR	3013	X'00000BC5'
MQRCCF_CFIN_PARM_ID_ERROR	3014	X'00000BC6'
MQRCCF_CFST_PARM_ID_ERROR	3015	X'00000BC7'
MQRCCF_MSG_LENGTH_ERROR	3016	X'00000BC8'
MQRCCF_CFIN_DUPLICATE_PARM	3017	X'00000BC9'
MQRCCF_CFST_DUPLICATE_PARM	3018	X'00000BCA'
MQRCCF_PARM_COUNT_TOO_SMALL	3019	X'00000BCB'
MQRCCF_PARM_COUNT_TOO_BIG	3020	X'00000BCC'
MQRCCF_Q_ALREADY_IN_CELL	3021	X'00000BCD'
MQRCCF_Q_TYPE_ERROR	3022	X'00000BCE'
MQRCCF_MD_FORMAT_ERROR	3023	X'00000BCF'
MQRCCF_CFSL_LENGTH_ERROR	3024	X'00000BD0'
MQRCCF_REPLACE_VALUE_ERROR	3025	X'00000BD1'
MQRCCF_CFIL_DUPLICATE_VALUE	3026	X'00000BD2'
MQRCCF_CFIL_COUNT_ERROR	3027	X'00000BD3'
MQRCCF_CFIL_LENGTH_ERROR	3028	X'00000BD4'
MQRCCF_QUIESCE_VALUE_ERROR	3029	X'00000BD5'
MQRCCF_MODE_VALUE_ERROR	3029	X'00000BD5'
MQRCCF_MSG_SEQ_NUMBER_ERROR	3030	X'00000BD6'
MQRCCF_PING_DATA_COUNT_ERROR	3031	X'00000BD7'
MQRCCF_PING_DATA_COMPARE_ERROR	3032	X'00000BD8'
MQRCCF_CFSL_PARM_ID_ERROR	3033	X'00000BD9'
MQRCCF_CHANNEL_TYPE_ERROR	3034	X'00000BDA'
MQRCCF_PARM_SEQUENCE_ERROR	3035	X'00000BDB'
MQRCCF_XMIT_PROTOCOL_TYPE_ERR	3036	X'00000BDC'
MQRCCF_BATCH_SIZE_ERROR	3037	X'00000BDD'
MQRCCF_DISC_INT_ERROR	3038	X'00000BDE'
MQRCCF_SHORT_RETRY_ERROR	3039	X'00000BDF'
MQRCCF_SHORT_TIMER_ERROR	3040	X'00000BE0'
MQRCCF_LONG_RETRY_ERROR	3041	X'00000BE1'
MQRCCF_LONG_TIMER_ERROR	3042	X'00000BE2'
MQRCCF_SEQ_NUMBER_WRAP_ERROR	3043	X'00000BE3'
MQRCCF_MAX_MSG_LENGTH_ERROR	3044	X'00000BE4'
MQRCCF_PUT_AUTH_ERROR	3045	X'00000BE5'
MQRCCF_PURGE_VALUE_ERROR	3046	X'00000BE6'
MQRCCF_CFIL_PARM_ID_ERROR	3047	X'00000BE7'

Table 306. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQRCCF_MSG_TRUNCATED	3048	X'00000BE8'
MQRCCF_CCSID_ERROR	3049	X'00000BE9'
MQRCCF_ENCODING_ERROR	3050	X'00000BEA'
MQRCCF_QUEUES_VALUE_ERROR	3051	X'00000BEB'
MQRCCF_DATA_CONV_VALUE_ERROR	3052	X'00000BEC'
MQRCCF_INDOUBT_VALUE_ERROR	3053	X'00000BED'
MQRCCF_ESCAPE_TYPE_ERROR	3054	X'00000BEE'
MQRCCF_REPOS_VALUE_ERROR	3055	X'00000BEF'
MQRCCF_CHANNEL_TABLE_ERROR	3062	X'00000BF6'
MQRCCF_MCA_TYPE_ERROR	3063	X'00000BF7'
MQRCCF_CHL_INST_TYPE_ERROR	3064	X'00000BF8'
MQRCCF_CHL_STATUS_NOT_FOUND	3065	X'00000BF9'
MQRCCF_CFSL_DUPLICATE_PARM	3066	X'00000BFA'
MQRCCF_CFSL_TOTAL_LENGTH_ERROR	3067	X'00000BFB'
MQRCCF_CFSL_COUNT_ERROR	3068	X'00000BFC'
MQRCCF_CFSL_STRING_LENGTH_ERR	3069	X'00000BFD'
MQRCCF_BROKER_DELETED	3070	X'00000BFE'
MQRCCF_STREAM_ERROR	3071	X'00000BFF'
MQRCCF_TOPIC_ERROR	3072	X'00000C00'
MQRCCF_NOT_REGISTERED	3073	X'00000C01'
MQRCCF_Q_MGR_NAME_ERROR	3074	X'00000C02'
MQRCCF_INCORRECT_STREAM	3075	X'00000C03'
MQRCCF_Q_NAME_ERROR	3076	X'00000C04'
MQRCCF_NO_RETAINED_MSG	3077	X'00000C05'
MQRCCF_DUPLICATE_IDENTITY	3078	X'00000C06'
MQRCCF_INCORRECT_Q	3079	X'00000C07'
MQRCCF_CORREL_ID_ERROR	3080	X'00000C08'
MQRCCF_NOT_AUTHORIZED	3081	X'00000C09'
MQRCCF_UNKNOWN_STREAM	3082	X'00000C0A'
MQRCCF_REG_OPTIONS_ERROR	3083	X'00000C0B'
MQRCCF_PUB_OPTIONS_ERROR	3084	X'00000C0C'
MQRCCF_UNKNOWN_BROKER	3085	X'00000C0D'
MQRCCF_Q_MGR_CCSID_ERROR	3086	X'00000C0E'
MQRCCF_DEL_OPTIONS_ERROR	3087	X'00000C0F'
MQRCCF_CLUSTER_NAME_CONFLICT	3088	X'00000C10'
MQRCCF_REPOS_NAME_CONFLICT	3089	X'00000C11'
MQRCCF_CLUSTER_Q_USAGE_ERROR	3090	X'00000C12'
MQRCCF_ACTION_VALUE_ERROR	3091	X'00000C13'
MQRCCF_COMMS_LIBRARY_ERROR	3092	X'00000C14'

Table 306. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRCCF_NETBIOS_NAME_ERROR	3093	X'00000C15'
MQRCCF_BROKER_COMMAND_FAILED	3094	X'00000C16'
MQRCCF_CFST_CONFLICTING_PARM	3095	X'00000C17'
MQRCCF_PATH_NOT_VALID	3096	X'00000C18'
MQRCCF_PARM_SYNTAX_ERROR	3097	X'00000C19'
MQRCCF_PWD_LENGTH_ERROR	3098	X'00000C1A'
MQRCCF_FILTER_ERROR	3150	X'00000C4E'
MQRCCF_WRONG_USER	3151	X'00000C4F'
MQRCCF_DUPLICATE_SUBSCRIPTION	3152	X'00000C50'
MQRCCF_SUB_NAME_ERROR	3153	X'00000C51'
MQRCCF_SUB_IDENTITY_ERROR	3154	X'00000C52'
MQRCCF_SUBSCRIPTION_IN_USE	3155	X'00000C53'
MQRCCF_SUBSCRIPTION_LOCKED	3156	X'00000C54'
MQRCCF_ALREADY_JOINED	3157	X'00000C55'
MQRCCF_OBJECT_IN_USE	3160	X'00000C58'
MQRCCF_UNKNOWN_FILE_NAME	3161	X'00000C59'
MQRCCF_FILE_NOT_AVAILABLE	3162	X'00000C5A'
MQRCCF_DISC_RETRY_ERROR	3163	X'00000C5B'
MQRCCF_ALLOC_RETRY_ERROR	3164	X'00000C5C'
MQRCCF_ALLOC_SLOW_TIMER_ERROR	3165	X'00000C5D'
MQRCCF_ALLOC_FAST_TIMER_ERROR	3166	X'00000C5E'
MQRCCF_PORT_NUMBER_ERROR	3167	X'00000C5F'
MQRCCF_CHL_SYSTEM_NOT_ACTIVE	3168	X'00000C60'
MQRCCF_ENTITY_NAME_MISSING	3169	X'00000C61'
MQRCCF_PROFILE_NAME_ERROR	3170	X'00000C62'
MQRCCF_AUTH_VALUE_ERROR	3171	X'00000C63'
MQRCCF_AUTH_VALUE_MISSING	3172	X'00000C64'
MQRCCF_OBJECT_TYPE_MISSING	3173	X'00000C65'
MQRCCF_CONNECTION_ID_ERROR	3174	X'00000C66'
MQRCCF_LOG_TYPE_ERROR	3175	X'00000C67'
MQRCCF_PROGRAM_NOT_AVAILABLE	3176	X'00000C68'
MQRCCF_PROGRAM_AUTH_FAILED	3177	X'00000C69'
MQRCCF_NONE_FOUND	3200	X'00000C80'
MQRCCF_SECURITY_SWITCH_OFF	3201	X'00000C81'
MQRCCF_SECURITY_REFRESH_FAILED	3202	X'00000C82'
MQRCCF_PARM_CONFLICT	3203	X'00000C83'
MQRCCF_COMMAND_INHIBITED	3204	X'00000C84'
MQRCCF_OBJECT_BEING_DELETED	3205	X'00000C85'
MQRCCF_STORAGE_CLASS_IN_USE	3207	X'00000C87'

Table 306. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQRCCF_OBJECT_NAME_RESTRICTED	3208	X'00000C88'
MQRCCF_OBJECT_LIMIT_EXCEEDED	3209	X'00000C89'
MQRCCF_OBJECT_OPEN_FORCE	3210	X'00000C8A'
MQRCCF_DISPOSITION_CONFLICT	3211	X'00000C8B'
MQRCCF_Q_MGR_NOT_IN_QSG	3212	X'00000C8C'
MQRCCF_ATTR_VALUE_FIXED	3213	X'00000C8D'
MQRCCF_NAMELIST_ERROR	3215	X'00000C8F'
MQRCCF_NO_CHANNEL_INITIATOR	3217	X'00000C91'
MQRCCF_CHANNEL_INITIATOR_ERROR	3218	X'00000C92'
MQRCCF_COMMAND_LEVEL_CONFLICT	3222	X'00000C96'
MQRCCF_Q_ATTR_CONFLICT	3223	X'00000C97'
MQRCCF_EVENTS_DISABLED	3224	X'00000C98'
MQRCCF_COMMAND_SCOPE_ERROR	3225	X'00000C99'
MQRCCF_COMMAND_REPLY_ERROR	3226	X'00000C9A'
MQRCCF_FUNCTION_RESTRICTED	3227	X'00000C9B'
MQRCCF_PARM_MISSING	3228	X'00000C9C'
MQRCCF_PARM_VALUE_ERROR	3229	X'00000C9D'
MQRCCF_COMMAND_LENGTH_ERROR	3230	X'00000C9E'
MQRCCF_COMMAND_ORIGIN_ERROR	3231	X'00000C9F'
MQRCCF_LISTENER_CONFLICT	3232	X'00000CA0'
MQRCCF_LISTENER_STARTED	3233	X'00000CA1'
MQRCCF_LISTENER_STOPPED	3234	X'00000CA2'
MQRCCF_CHANNEL_ERROR	3235	X'00000CA3'
MQRCCF_CF_STRUC_ERROR	3236	X'00000CA4'
MQRCCF_UNKNOWN_USER_ID	3237	X'00000CA5'
MQRCCF_UNEXPECTED_ERROR	3238	X'00000CA6'
MQRCCF_NO_XCF_PARTNER	3239	X'00000CA7'
MQRCCF_CFGR_PARM_ID_ERROR	3240	X'00000CA8'
MQRCCF_CFIF_LENGTH_ERROR	3241	X'00000CA9'
MQRCCF_CFIF_OPERATOR_ERROR	3242	X'00000CAA'
MQRCCF_CFIF_PARM_ID_ERROR	3243	X'00000CAB'
MQRCCF_CFSF_FILTER_VAL_LEN_ERR	3244	X'00000CAC'
MQRCCF_CFSF_LENGTH_ERROR	3245	X'00000CAD'
MQRCCF_CFSF_OPERATOR_ERROR	3246	X'00000CAE'
MQRCCF_CFSF_PARM_ID_ERROR	3247	X'00000CAF'
MQRCCF_TOO_MANY_FILTERS	3248	X'00000CB0'
MQRCCF_LISTENER_RUNNING	3249	X'00000CB1'
MQRCCF_LSTR_STATUS_NOT_FOUND	3250	X'00000CB2'
MQRCCF_SERVICE_RUNNING	3251	X'00000CB3'

Table 306. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRCCF_SERV_STATUS_NOT_FOUND	3252	X'00000CB4'
MQRCCF_SERVICE_STOPPED	3253	X'00000CB5'
MQRCCF_CFBS_DUPLICATE_PARM	3254	X'00000CB6'
MQRCCF_CFBS_LENGTH_ERROR	3255	X'00000CB7'
MQRCCF_CFBS_PARM_ID_ERROR	3256	X'00000CB8'
MQRCCF_CFBS_STRING_LENGTH_ERR	3257	X'00000CB9'
MQRCCF_CFGR_LENGTH_ERROR	3258	X'00000CBA'
MQRCCF_CFGR_PARM_COUNT_ERROR	3259	X'00000CBB'
MQRCCF_CONN_NOT_STOPPED	3260	X'00000CBC'
MQRCCF_SERVICE_REQUEST_PENDING	3261	X'00000CBD'
MQRCCF_NO_START_CMD	3262	X'00000CBE'
MQRCCF_NO_STOP_CMD	3263	X'00000CBF'
MQRCCF_CFBF_LENGTH_ERROR	3264	X'00000CC0'
MQRCCF_CFBF_PARM_ID_ERROR	3265	X'00000CC1'
MQRCCF_CFBF_OPERATOR_ERROR	3266	X'00000CC2'
MQRCCF_CFBF_FILTER_VAL_LEN_ERR	3267	X'00000CC3'
MQRCCF_LISTENER_STILL_ACTIVE	3268	X'00000CC4'
MQRCCF_DEF_XMIT_Q_CLUS_ERROR	3269	X'00000CC5'
MQRCCF_TOPICSTR_ALREADY_EXISTS	3300	X'00000CE4'
MQRCCF_SHARING_CONVS_ERROR	3301	X'00000CE5'
MQRCCF_SHARING_CONVS_TYPE	3302	X'00000CE6'
MQRCCF_SECURITY_CASE_CONFLICT	3303	X'00000CE7'
MQRCCF_TOPIC_TYPE_ERROR	3305	X'00000CE9'
MQRCCF_MAX_INSTANCES_ERROR	3306	X'00000CEA'
MQRCCF_MAX_INSTS_PER_CLNT_ERR	3307	X'00000CEB'
MQRCCF_TOPIC_STRING_NOT_FOUND	3308	X'00000CEC'
MQRCCF_SUBSCRIPTION_POINT_ERR	3309	X'00000CED'
MQRCCF_SUB_ALREADY_EXISTS	3311	X'00000CEF'
MQRCCF_UNKNOWN_OBJECT_NAME	3312	X'00000CF0'
MQRCCF_REMOTE_Q_NAME_ERROR	3313	X'00000CF1'
MQRCCF_DURABILITY_NOT_ALLOWED	3314	X'00000CF2'
MQRCCF_HOBJ_ERROR	3315	X'00000CF3'
MQRCCF_DEST_NAME_ERROR	3316	X'00000CF4'
MQRCCF_INVALID_DESTINATION	3317	X'00000CF5'
MQRCCF_PUBSUB_INHIBITED	3318	X'00000CF6'
MQRCCF_CHLAUTH_TYPE_ERROR	3326	X'00000CFE'
MQRCCF_CHLAUTH_ACTION_ERROR	3327	X'00000CFF'
MQRCCF_CHLAUTH_USERSRC_ERROR	3335	X'00000D07'
MQRCCF_WRONG_CHLAUTH_TYPE	3336	X'00000D08'

Table 306. Values of constants (continued)

Name	Decimal value	Hexadecimal value
MQRCCF_CHLAUTH_ALREADY_EXISTS	3337	X'00000D09'
MQRCCF_CHLAUTH_NOT_FOUND	3338	X'00000D0A'
MQRCCF_WRONG_CHLAUTH_ACTION	3339	X'00000D0B'
MQRCCF_WRONG_CHLAUTH_USERSRC	3340	X'00000D0C'
MQRCCF_CHLAUTH_WARN_ERROR	3341	X'00000D0D'
MQRCCF_WRONG_CHLAUTH_MATCH	3342	X'00000D0E'
MQRCCF_IPADDR_RANGE_CONFLICT	3343	X'00000D0F'
MQRCCF_CHLAUTH_MAX_EXCEEDED	3344	X'00000D10'
MQRCCF_IPADDR_ERROR	3345	X'00000D11'
MQRCCF_IPADDR_RANGE_ERROR	3346	X'00000D12'
MQRCCF_PROFILE_NAME_MISSING	3347	X'00000D13'
MQRCCF_CHLAUTH_CLNTUSER_ERROR	3348	X'00000D14'
MQRCCF_CHLAUTH_NAME_ERROR	3349	X'00000D15'
MQRCCF_SUITE_B_ERROR	3353	X'00000D19'
MQRCCF_PSCLUS_DISABLED_TOPDEF	3359	X'00000D1F'
MQRCCF_PSCLUS_TOPIC_EXISTS	3360	X'00000D20'
MQRCCF_INVALID_PROTOCOL	3365	X'00000D25'
MQRCCF_OBJECT_ALREADY_EXISTS	4001	X'00000FA1'
MQRCCF_OBJECT_WRONG_TYPE	4002	X'00000FA2'
MQRCCF_LIKE_OBJECT_WRONG_TYPE	4003	X'00000FA3'
MQRCCF_OBJECT_OPEN	4004	X'00000FA4'
MQRCCF_ATTR_VALUE_ERROR	4005	X'00000FA5'
MQRCCF_UNKNOWN_Q_MGR	4006	X'00000FA6'
MQRCCF_Q_WRONG_TYPE	4007	X'00000FA7'
MQRCCF_OBJECT_NAME_ERROR	4008	X'00000FA8'
MQRCCF_ALLOCATE_FAILED	4009	X'00000FA9'
MQRCCF_HOST_NOT_AVAILABLE	4010	X'00000FAA'
MQRCCF_CONFIGURATION_ERROR	4011	X'00000FAB'
MQRCCF_CONNECTION_REFUSED	4012	X'00000FAC'
MQRCCF_ENTRY_ERROR	4013	X'00000FAD'
MQRCCF_SEND_FAILED	4014	X'00000FAE'
MQRCCF_RECEIVED_DATA_ERROR	4015	X'00000FAF'
MQRCCF_RECEIVE_FAILED	4016	X'00000FB0'
MQRCCF_CONNECTION_CLOSED	4017	X'00000FB1'
MQRCCF_NO_STORAGE	4018	X'00000FB2'
MQRCCF_NO_COMMS_MANAGER	4019	X'00000FB3'
MQRCCF_LISTENER_NOT_STARTED	4020	X'00000FB4'
MQRCCF_BIND_FAILED	4024	X'00000FB8'
MQRCCF_CHANNEL_INDOUBT	4025	X'00000FB9'

Table 306. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRCCF_MQCONN_FAILED	4026	X'00000FBA'
MQRCCF_MQOPEN_FAILED	4027	X'00000FBB'
MQRCCF_MQGET_FAILED	4028	X'00000FBC'
MQRCCF_MQPUT_FAILED	4029	X'00000FBD'
MQRCCF_PING_ERROR	4030	X'00000FBE'
MQRCCF_CHANNEL_IN_USE	4031	X'00000FBF'
MQRCCF_CHANNEL_NOT_FOUND	4032	X'00000FC0'
MQRCCF_UNKNOWN_REMOTE_CHANNEL	4033	X'00000FC1'
MQRCCF_REMOTE_QM_UNAVAILABLE	4034	X'00000FC2'
MQRCCF_REMOTE_QM_TERMINATING	4035	X'00000FC3'
MQRCCF_MQINQ_FAILED	4036	X'00000FC4'
MQRCCF_NOT_XMIT_Q	4037	X'00000FC5'
MQRCCF_CHANNEL_DISABLED	4038	X'00000FC6'
MQRCCF_USER_EXIT_NOT_AVAILABLE	4039	X'00000FC7'
MQRCCF_COMMIT_FAILED	4040	X'00000FC8'
MQRCCF_WRONG_CHANNEL_TYPE	4041	X'00000FC9'
MQRCCF_CHANNEL_ALREADY_EXISTS	4042	X'00000FCA'
MQRCCF_DATA_TOO_LARGE	4043	X'00000FCB'
MQRCCF_CHANNEL_NAME_ERROR	4044	X'00000FCC'
MQRCCF_XMIT_Q_NAME_ERROR	4045	X'00000FCD'
MQRCCF_MCA_NAME_ERROR	4047	X'00000FCF'
MQRCCF_SEND_EXIT_NAME_ERROR	4048	X'00000FDD'
MQRCCF_SEC_EXIT_NAME_ERROR	4049	X'00000FDD'
MQRCCF_MSG_EXIT_NAME_ERROR	4050	X'00000FDD'
MQRCCF_RCV_EXIT_NAME_ERROR	4051	X'00000FDD'
MQRCCF_XMIT_Q_NAME_WRONG_TYPE	4052	X'00000FDD'
MQRCCF_MCA_NAME_WRONG_TYPE	4053	X'00000FDD'
MQRCCF_DISC_INT_WRONG_TYPE	4054	X'00000FDD'
MQRCCF_SHORT_RETRY_WRONG_TYPE	4055	X'00000FDD'
MQRCCF_SHORT_TIMER_WRONG_TYPE	4056	X'00000FDD'
MQRCCF_LONG_RETRY_WRONG_TYPE	4057	X'00000FDD'
MQRCCF_LONG_TIMER_WRONG_TYPE	4058	X'00000FDD'
MQRCCF_PUT_AUTH_WRONG_TYPE	4059	X'00000FDD'
MQRCCF_KEEP_ALIVE_INT_ERROR	4060	X'00000FDD'
MQRCCF_MISSING_CONN_NAME	4061	X'00000FDD'
MQRCCF_CONN_NAME_ERROR	4062	X'00000FDD'
MQRCCF_MQSET_FAILED	4063	X'00000FDD'
MQRCCF_CHANNEL_NOT_ACTIVE	4064	X'00000FDD'
MQRCCF_TERMINATED_BY_SEC_EXIT	4065	X'00000FDD'

Table 306. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRCCF_DYNAMIC_Q_SCOPE_ERROR	4067	X'00000FE3'
MQRCCF_CELL_DIR_NOT_AVAILABLE	4068	X'00000FE4'
MQRCCF_MR_COUNT_ERROR	4069	X'00000FE5'
MQRCCF_MR_COUNT_WRONG_TYPE	4070	X'00000FE6'
MQRCCF_MR_EXIT_NAME_ERROR	4071	X'00000FE7'
MQRCCF_MR_EXIT_NAME_WRONG_TYPE	4072	X'00000FE8'
MQRCCF_MR_INTERVAL_ERROR	4073	X'00000FE9'
MQRCCF_MR_INTERVAL_WRONG_TYPE	4074	X'00000FEA'
MQRCCF_NPM_SPEED_ERROR	4075	X'00000FEB'
MQRCCF_NPM_SPEED_WRONG_TYPE	4076	X'00000FEC'
MQRCCF_HB_INTERVAL_ERROR	4077	X'00000FED'
MQRCCF_HB_INTERVAL_WRONG_TYPE	4078	X'00000FEE'
MQRCCF_CHAD_ERROR	4079	X'00000FEF'
MQRCCF_CHAD_WRONG_TYPE	4080	X'00000FF0'
MQRCCF_CHAD_EVENT_ERROR	4081	X'00000FF1'
MQRCCF_CHAD_EVENT_WRONG_TYPE	4082	X'00000FF2'
MQRCCF_CHAD_EXIT_ERROR	4083	X'00000FF3'
MQRCCF_CHAD_EXIT_WRONG_TYPE	4084	X'00000FF4'
MQRCCF_SUPPRESSED_BY_EXIT	4085	X'00000FF5'
MQRCCF_BATCH_INT_ERROR	4086	X'00000FF6'
MQRCCF_BATCH_INT_WRONG_TYPE	4087	X'00000FF7'
MQRCCF_NET_PRIORITY_ERROR	4088	X'00000FF8'
MQRCCF_NET_PRIORITY_WRONG_TYPE	4089	X'00000FF9'
MQRCCF_CHANNEL_CLOSED	4090	X'00000FFA'
MQRCCF_Q_STATUS_NOT_FOUND	4091	X'00000FFB'
MQRCCF_SSL_CIPHER_SPEC_ERROR	4092	X'00000FFC'
MQRCCF_SSL_PEER_NAME_ERROR	4093	X'00000FFD'
MQRCCF_SSL_CLIENT_AUTH_ERROR	4094	X'00000FFE'
MQRCCF_RETAINED_NOT_SUPPORTED	4095	X'00000FFF'

MQRCN_* (Client reconnect Constants)

Table 307. Values of constants		
Name	Decimal value	Hexadecimal value
MQRCN_NO	0	X'00000000'
MQRCN_YES	1	X'00000001'
MQRCN_Q_MGR	2	X'00000002'
MQRCN_DISABLED	3	X'00000003'

MQRCVTIME_* (Receive Timeout Types)

Table 308. Values of constants		
Name	Decimal value	Hexadecimal value
MQRCVTIME_MULTIPLY	0	X'00000000'
MQRCVTIME_ADD	1	X'00000001'
MQRCVTIME_EQUAL	2	X'00000002'

MQREADA_* (Read Ahead Values)

Table 309. Values of constants		
Name	Decimal value	Hexadecimal value
MQREADA_NO	0	X'00000000'
MQREADA_YES	1	X'00000001'
MQREADA_DISABLED	2	X'00000002'
MQREADA_INHIBITED	3	X'00000003'
MQREADA_BACKLOG	4	X'00000004'

MQRECORDING_* (Recording Options)

Table 310. Values of constants		
Name	Decimal value	Hexadecimal value
MQRECORDING_DISABLED	0	X'00000000'
MQRECORDING_Q	1	X'00000001'
MQRECORDING_MSG	2	X'00000002'

MQREGO_* (Publish/Subscribe Registration Options)

Table 311. Values of constants		
Name	Decimal value	Hexadecimal value
MQREGO_NONE	0	X'00000000'
MQREGO_CORREL_ID_AS_IDENTITY	1	X'00000001'
MQREGO_ANONYMOUS	2	X'00000002'
MQREGO_LOCAL	4	X'00000004'
MQREGO_DIRECT_REQUESTS	8	X'00000008'
MQREGO_NEW_PUBLICATIONS_ONLY	16	X'00000010'
MQREGO_PUBLISH_ON_REQUEST_ONLY	32	X'00000020'
MQREGO_DEREGISTER_ALL	64	X'00000040'
MQREGO_INCLUDE_STREAM_NAME	128	X'00000080'
MQREGO_INFORM_IF_RETAINED	256	X'00000100'
MQREGO_DUPLICATES_OK	512	X'00000200'
MQREGO_NON_PERSISTENT	1024	X'00000400'
MQREGO_PERSISTENT	2048	X'00000800'
MQREGO_PERSISTENT_AS_PUBLISH	4096	X'00001000'
MQREGO_PERSISTENT_AS_Q	8192	X'00002000'

Table 311. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQREGO_ADD_NAME	16384	X'00004000'
MQREGO_NO_ALTERATION	32768	X'00008000'
MQREGO_FULL_RESPONSE	65536	X'00010000'
MQREGO_JOIN_SHARED	131072	X'00020000'
MQREGO_JOIN_EXCLUSIVE	262144	X'00040000'
MQREGO_LEAVE_ONLY	524288	X'00080000'
MQREGO_VARIABLE_USER_ID	1048576	X'00100000'
MQREGO_LOCKED	2097152	X'00200000'

MQRFH_* (Rules and formatting header structure and Flags)

Rules and formatting header structure

Table 312. Structures of constants	
Name	Structure
MQRFH_STRUC_ID	"RFH↵"
MQRFH_STRUC_ID_ARRAY	'R','F','H','↵'

Note: The symbol ↵ represents a single blank character.

Table 313. Values of constants		
Name	Decimal value	Hexadecimal value
MQRFH_VERSION_1	1	X'00000001'
MQRFH_VERSION_2	2	X'00000002'
MQRFH_STRUC_LENGTH_FIXED	32	X'00000020'
MQRFH_STRUC_LENGTH_FIXED_2	36	X'00000024'

Rules and formatting header Flags

Table 314. Values of constants		
Name	Decimal value	Hexadecimal value
MQRFH_NONE	0	X'00000000'
MQRFH_NO_FLAGS	0	X'00000000'

MQRFH2_* (Publish/Subscribe Options Tag RFH2 Top-level folder Tags)

Table 315. Values of constants		
Name	Decimal value	Hexadecimal value
MQRFH2_NAME_VALUE_VERSION	1	X'00000001'

MQRFH2_* (Publish/Subscribe Options Tag Tag names)

MQRFH2_PUBSUB_CMD_FOLDER	"psc"
MQRFH2_PUBSUB_RESP_FOLDER	"pscr"

MQRFH2_MSG_CONTENT_FOLDER	"mcd"
MQRFH2_USER_FOLDER	"usr"

MQRFH2_* (Publish/Subscribe Options Tag XML tag names)

MQRFH2_PUBSUB_CMD_FOLDER_B	"<psc>"
MQRFH2_PUBSUB_CMD_FOLDER_E	"</psc>"
MQRFH2_PUBSUB_RESP_FOLDER_B	"<pscr>"
MQRFH2_PUBSUB_RESP_FOLDER_E	"</pscr>"
MQRFH2_MSG_CONTENT_FOLDER_B	"<mcd>"
MQRFH2_MSG_CONTENT_FOLDER_E	"</mcd>"
MQRFH2_USER_FOLDER_B	"<usr>"
MQRFH2_USER_FOLDER_E	"</usr>"

MQRL_* (Returned Length)

<i>Table 316. Values of constants</i>		
Name	Decimal value	Hexadecimal value
MQRL_UNDEFINED	-1	X'FFFFFFFF'

MQRMH_* (Reference message header structure)

<i>Table 317. Structures of constants</i>	
Name	Structure
MQRMH_STRUC_ID	"RMH↯"
MQRMH_STRUC_ID_ARRAY	'R', 'M', 'H', '↯'

Note: The symbol ↯ represents a single blank character.

<i>Table 318. Values of constants</i>		
Name	Decimal value	Hexadecimal value
MQRMH_VERSION_1	1	X'00000001'
MQRMH_CURRENT_VERSION	1	X'00000001'

MQRMHF_* (Reference message header Flags)

<i>Table 319. Values of constants</i>		
Name	Decimal value	Hexadecimal value
MQRMHF_LAST	1	X'00000001'
MQRMHF_NOT_LAST	0	X'00000000'

MQRO_* (Report Options)

<i>Table 320. Values of constants</i>		
Name	Decimal value	Hexadecimal value
MQRO_EXCEPTION	16777216	X'01000000'

Table 320. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRO_EXCEPTION_WITH_DATA	50331648	X'03000000'
MQRO_EXCEPTION_WITH_FULL_DATA	117440512	X'07000000'
MQRO_EXPIRATION	2097152	X'00200000'
MQRO_EXPIRATION_WITH_DATA	6291456	X'00600000'
MQRO_EXPIRATION_WITH_FULL_DATA	14680064	X'00E00000'
MQRO_COA	256	X'00000100'
MQRO_COA_WITH_DATA	768	X'00000300'
MQRO_COA_WITH_FULL_DATA	1792	X'00000700'
MQRO_COD	2048	X'00000800'
MQRO_COD_WITH_DATA	6144	X'00001800'
MQRO_COD_WITH_FULL_DATA	14336	X'00003800'
MQRO_PAN	1	X'00000001'
MQRO_NAN	2	X'00000002'
MQRO_ACTIVITY	4	X'00000004'
MQRO_NEW_MSG_ID	0	X'00000000'
MQRO_PASS_MSG_ID	128	X'00000080'
MQRO_COPY_MSG_ID_TO_CORREL_ID	0	X'00000000'
MQRO_PASS_CORREL_ID	64	X'00000040'
MQRO_DEAD_LETTER_Q	0	X'00000000'
MQRO_DISCARD_MSG	134217728	X'08000000'
MQRO_PASS_DISCARD_AND_EXPIRY	16384	X'00004000'
MQRO_NONE	0	X'00000000'

MQRO_* (Report Options Masks)

Table 321. Values of constants		
Name	Decimal value	Hexadecimal value
MQRO_REJECT_UNSUP_MASK	270270464	X'101C0000'
MQRO_ACCEPT_UNSUP_MASK	-270532353	X'EFE000FF'
MQRO_ACCEPT_UNSUP_IF_XMIT_MASK	261888	X'0003FF00'

MQROUTE_* (Trace-route)

Trace-route Max Activities (MQIACF_MAX_ACTIVITIES)

Table 322. Values of constants		
Name	Decimal value	Hexadecimal value
MQROUTE_UNLIMITED_ACTIVITIES	0	X'00000000'

Trace-route Detail (MQIACF_ROUTE_DETAIL)

Table 323. Values of constants		
Name	Decimal value	Hexadecimal value
MQROUTE_DETAIL_LOW	2	X'00000002'
MQROUTE_DETAIL_MEDIUM	8	X'00000008'
MQROUTE_DETAIL_HIGH	32	X'00000020'

Trace-route Forwarding (MQIACF_ROUTE_FORWARDING)

Table 324. Values of constants		
Name	Decimal value	Hexadecimal value
MQROUTE_FORWARD_ALL	256	X'00000100'
MQROUTE_FORWARD_IF_SUPPORTED	512	X'00000200'
MQROUTE_FORWARD_REJ_UNSUP_MASK	-65536	X'FFFF0000'

Trace-route Delivery (MQIACF_ROUTE_DELIVERY)

Table 325. Values of constants		
Name	Decimal value	Hexadecimal value
MQROUTE_DELIVER_YES	4096	X'00001000'
MQROUTE_DELIVER_NO	8192	X'00002000'
MQROUTE_DELIVER_REJ_UNSUP_MASK	-65536	X'FFFF0000'

Trace-route Accumulation (MQIACF_ROUTE_ACCUMULATION)

Table 326. Values of constants		
Name	Decimal value	Hexadecimal value
MQROUTE_ACCUMULATE_NONE	65539	X'00010003'
MQROUTE_ACCUMULATE_IN_MSG	65540	X'00010004'
MQROUTE_ACCUMULATE_AND_REPLY	65541	X'00010005'

MQRP_* (Command format Replace Options)

Table 327. Values of constants		
Name	Decimal value	Hexadecimal value
MQRP_YES	1	X'00000001'
MQRP_NO	0	X'00000000'

MQRQ_* (Command format Reason Qualifiers)

Table 328. Values of constants		
Name	Decimal value	Hexadecimal value
MQRQ_CONN_NOT_AUTHORIZED	1	X'00000001'
MQRQ_OPEN_NOT_AUTHORIZED	2	X'00000002'
MQRQ_CLOSE_NOT_AUTHORIZED	3	X'00000003'
MQRQ_CMD_NOT_AUTHORIZED	4	X'00000004'

Table 328. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQRQ_Q_MGR_STOPPING	5	X'00000005'
MQRQ_Q_MGR QUIESCING	6	X'00000006'
MQRQ_CHANNEL_STOPPED_OK	7	X'00000007'
MQRQ_CHANNEL_STOPPED_ERROR	8	X'00000008'
MQRQ_CHANNEL_STOPPED_RETRY	9	X'00000009'
MQRQ_CHANNEL_STOPPED_DISABLED	10	X'0000000A'
MQRQ_BRIDGE_STOPPED_OK	11	X'0000000B'
MQRQ_BRIDGE_STOPPED_ERROR	12	X'0000000C'
MQRQ_SSL_HANDSHAKE_ERROR	13	X'0000000D'
MQRQ_SSL_CIPHER_SPEC_ERROR	14	X'0000000E'
MQRQ_SSL_CLIENT_AUTH_ERROR	15	X'0000000F'
MQRQ_SSL_PEER_NAME_ERROR	16	X'00000010'
MQRQ_SUB_NOT_AUTHORIZED	17	X'00000011'
MQRQ_SUB_DEST_NOT_AUTHORIZED	18	X'00000012'
MQRQ_SSL_UNKNOWN_REVOCATION	19	X'00000013'
MQRQ_SYS_CONN_NOT_AUTHORIZED	20	X'00000014'
MQRQ_CHANNEL_BLOCKED_ADDRESS	21	X'00000015'
MQRQ_CHANNEL_BLOCKED_USERID	22	X'00000016'
MQRQ_CHANNEL_BLOCKED_NOACCESS	23	X'00000017'
MQRQ_MAX_ACTIVE_CHANNELS	24	X'00000018'
MQRQ_MAX_CHANNELS	25	X'00000019'
MQRQ_SVRCONN_INST_LIMIT	26	X'0000001A'
MQRQ_CLIENT_INST_LIMIT!	27	X'0000001B'
MQRQ_CAF_NOT_INSTALLED	28	X'0000001C'

MQRT_* (Command format Refresh Types)

Table 329. Values of constants		
Name	Decimal value	Hexadecimal value
MQRT_CONFIGURATION	1	X'00000001'
MQRT_EXPIRY	2	X'00000002'
MQRT_NSPROC	3	X'00000003'
MQRT_PROXYSUB	4	X'00000004'

MQRU_* (Request Only)

Table 330. Values of constants		
Name	Decimal value	Hexadecimal value
MQRU_PUBLISH_ON_REQUEST	1	X'00000001'
MQRU_PUBLISH_ALL	2	X'00000002'

MQSCA_* (SSL Client Authentication)

Table 331. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCA_REQUIRED	0	X'00000000'
MQSCA_OPTIONAL	1	X'00000001'

MQSCO_* (SSL configuration options)

SSL configuration options structure

Table 332. Structures of constants	
Name	Structure
MQSCO_STRUC_ID	"SCO↵"
MQSCO_STRUC_ID_ARRAY	'S','C','O','↵'

Note: The symbol ↵ represents a single blank character.

Table 333. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCO_VERSION_1	1	X'00000001'
MQSCO_VERSION_2	2	X'00000002'
MQSCO_VERSION_3	3	X'00000003'
MQSCO_VERSION_4	4	X'00000004'
MQSCO_CURRENT_VERSION	4	X'00000004'

Note: The symbol ↵ represents a single blank character.

SSL configuration options Key Reset Count

Table 334. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCO_RESET_COUNT_DEFAULT	0	X'00000000'

Command format Queue Definition Scope

Table 335. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCO_Q_MGR	1	X'00000001'
MQSCO_CELL	2	X'00000002'

MQSCOPE_* (Publish scope)

Table 336. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCOPE_ALL	0	X'00000000'
MQSCOPE_AS_PARENT	1	X'00000001'
MQSCOPE_QMGR	4	X'00000004'

MQSCYC_* (Security Case)

Table 337. Values of constants		
Name	Decimal value	Hexadecimal value
MQSCYC_UPPER	0	X'00000000'
MQSCYC_MIXED	1	X'00000001'

MQSD_* (Object descriptor structure)

Table 338. Constant names and structures	
Name	Structure
MQSD_STRUC_ID	"SD¬¬"
MQSD_STRUC_ID_ARRAY	'S','D','¬','¬'

Note: The symbol ¬ represents a single blank character.

Table 339. Values of constants		
Name	Decimal value	Hexadecimal value
MQSD_VERSION_1	1	X'00000001'
MQSD_CURRENT_VERSION	1	X'00000001'

MQSECITEM_* (Command format Security Items)

Table 340. Values of constants		
Name	Decimal value	Hexadecimal value
MQSECITEM_ALL	0	X'00000000'
MQSECITEM_MQADMIN	1	X'00000001'
MQSECITEM_MQNLIST	2	X'00000002'
MQSECITEM_MQPROC	3	X'00000003'
MQSECITEM_MQQUEUE	4	X'00000004'
MQSECITEM_MQCONN	5	X'00000005'
MQSECITEM_MQCMLS	6	X'00000006'
MQSECITEM_MXADMIN	7	X'00000007'
MQSECITEM_MXNLIST	8	X'00000008'
MQSECITEM_MXPROC	9	X'00000009'
MQSECITEM_MXQUEUE	10	X'0000000A'
MQSECITEM_MXTOPIC	11	X'0000000B'

MQSECSW_* (Command format Security Switches and Switch States)

Command format Security Switches

Table 341. Values of constants		
Name	Decimal value	Hexadecimal value
MQSECSW_PROCESS	1	X'00000001'
MQSECSW_NAMELIST	2	X'00000002'
MQSECSW_Q	3	X'00000003'

Table 341. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQSECSW_TOPIC	4	X'00000004'
MQSECSW_CONTEXT	6	X'00000006'
MQSECSW_ALTERNATE_USER	7	X'00000007'
MQSECSW_COMMAND	8	X'00000008'
MQSECSW_CONNECTION	9	X'00000009'
MQSECSW_SUBSYSTEM	10	X'0000000A'
MQSECSW_COMMAND_RESOURCES	11	X'0000000B'
MQSECSW_Q_MGR	15	X'0000000F'
MQSECSW_QSG	16	X'00000010'

Command format Security Switch States

Table 342. Values of constants		
Name	Decimal value	Hexadecimal value
MQSECSW_OFF_FOUND	21	X'00000015'
MQSECSW_ON_FOUND	22	X'00000016'
MQSECSW_OFF_NOT_FOUND	23	X'00000017'
MQSECSW_ON_NOT_FOUND	24	X'00000018'
MQSECSW_OFF_ERROR	25	X'00000019'
MQSECSW_ON_OVERRIDDEN	26	X'0000001A'

MQSECTYPE_* (Command format Security Types)

Table 343. Values of constants		
Name	Decimal value	Hexadecimal value
MQSECTYPE_AUTHSERV	1	X'00000001'
MQSECTYPE_SSL	2	X'00000002'
MQSECTYPE_CLASSES	3	X'00000003'

MQSEG_* (Segmentation)

Table 344. Constant names and values	
Name	Value
MQSEG_INHIBITED	'␣'
MQSEG_ALLOWED	'A'

Note: The symbol ␣ represents a single blank character.

MQSEL_* (Special Selector Values)

Table 345. Values of constants		
Name	Decimal value	Hexadecimal value
MQSEL_ANY_SELECTOR	-30001	X'FFFF8ACF'
MQSEL_ANY_USER_SELECTOR	-30002	X'FFFF8ACE'

Table 345. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQSEL_ANY_SYSTEM_SELECTOR	-30003	X'FFFF8ACD'
MQSEL_ALL_SELECTORS	-30001	X'FFFF8ACF'
MQSEL_ALL_USER_SELECTORS	-30002	X'FFFF8ACE'
MQSEL_ALL_SYSTEM_SELECTORS	-30003	X'FFFF8ACD'

MQSELTYPE_* (Selector Types)

Table 346. Values of constants		
Name	Decimal value	Hexadecimal value
MQSELTYPE_NONE	0	X'00000000'
MQSELTYPE_STANDARD	1	X'00000001'
MQSELTYPE_EXTENDED	2	X'00000002'

MQSID_* (Security Identifier)

Table 347. Constant names and values	
Name	Value
MQSID_NONE	X'00...00' (40 nulls)
MQSID_NONE_ARRAY	'\0', '\0', ... (40 nulls)

MQSIDT_* (Security Identifier Types)

Table 348. Constant names and values	
Name	Hexadecimal value
MQSIDT_NONE	X'00'
MQSIDT_NT_SECURITY_ID	X'01'
MQSIDT_WAS_SECURITY_ID	X'02'

MQSIDT_NONE	X'00'
MQSIDT_NT_SECURITY_ID	X'01'
MQSIDT_WAS_SECURITY_ID	X'02'

MQSMPO_* (Set message property options and structure)

Set message property options structure

Table 349. Structures of constants	
Name	Structure
MQSMPO_STRUC_ID	"SMPO"
MQSMPO_STRUC_ID_ARRAY	'S', 'M', 'P', 'O'

Note: The symbol ↵ represents a single blank character.

Table 350. Values of constants		
Name	Decimal value	Hexadecimal value
MQSMPO_VERSION_1	1	X'00000001'
MQSMPO_CURRENT_VERSION	1	X'00000001'

Set Message Property Options

Table 351. Values of constants		
Name	Decimal value	Hexadecimal value
MQSMPO_SET_FIRST	0	X'00000000'
MQSMPO_SET_PROP_UNDER_CURSOR	1	X'00000001'
MQSMPO_SET_PROP_AFTER_CURSOR	2	X'00000002'
MQSMPO_APPEND_PROPERTY	4	X'00000004'
MQSMPO_SET_PROP_BEFORE_CURSOR	8	X'00000008'
MQSMPO_NONE	0	X'00000000'

MQSO_* (Subscribe Options)

Table 352. Values of constants		
Name	Decimal value	Hexadecimal value
MQSO_NONE	0	X'00000000'
MQSO_NON_DURABLE	0	X'00000000'
MQSO_READ_AHEAD_AS_Q_DEF	0	X'00000000'
MQSO_ALTER	1	X'00000001'
MQSO_CREATE	2	X'00000002'
MQSO_RESUME	4	X'00000004'
MQSO_DURABLE	8	X'00000008'
MQSO_GROUP_SUB	16	X'00000010'
MQSO_MANAGED	32	X'00000020'
MQSO_SET_IDENTITY_CONTEXT	64	X'00000040'
MQSO_FIXED_USERID	256	X'00000100'
MQSO_ANY_USERID	512	X'00000200'
MQSO_PUBLICATIONS_ON_REQUEST	2048	X'00000800'
MQSO_NEW_PUBLICATIONS_ONLY	4096	X'00001000'
MQSO_FAIL_IF QUIESCING	8192	X'00002000'
MQSO_ALTERNATE_USER_AUTHORITY	262144	X'00040000'
MQSO_WILDCARD_CHAR	1048576	X'00100000'
MQSO_WILDCARD_TOPIC	2097152	X'00200000'
MQSO_SET_CORREL_ID	4194304	X'00400000'
MQSO_SCOPE_QMGR	67108864	X'04000000'
MQSO_NO_READ_AHEAD	134217728	X'08000000'
MQSO_READ_AHEAD	268435456	X'10000000'

MQSP_* (Sync point Availability)

Table 353. Values of constants		
Name	Decimal value	Hexadecimal value
MQSP_AVAILABLE	1	X'00000001'
MQSP_NOT_AVAILABLE	0	X'00000000'

MQSQQM_* (Shared Queue Queue Manager Name)

Table 354. Values of constants		
Name	Decimal value	Hexadecimal value
MQSQQM_USE	0	X'00000000'
MQSQQM_IGNORE	1	X'00000001'

MQSR_* (Action)

Table 355. Values of constants		
Name	Decimal value	Hexadecimal value
MQSR_ACTION_PUBLICATION	1	X'00000001'

MQSRO_* (Subscription request options structure)

Table 356. Structures of constants	
Name	Structure
MQSRO_STRUC_ID	"SRO¬"
MQSRO_STRUC_ID_ARRAY	'S','R','O','¬'

Note: The symbol ¬ represents a single blank character.

Table 357. Values of constants		
Name	Decimal value	Hexadecimal value
MQSRO_VERSION_1	1	X'00000001'
MQSRO_CURRENT_VERSION	1	X'00000001'
MQSRO_NONE	0	X'00000000'
MQSRO_FAIL_IF_QUIESCING	8192	X'00002000'

MQSS_* (Segment Status)

Table 358. Constant names and structures	
Name	Structure
MQSS_NOT_A_SEGMENT	'¬'
MQSS_SEGMENT	'S'
MQSS_LAST_SEGMENT	'L'

Note: The symbol ¬ represents a single blank character.

MQSSL_* (SSL FIPS Requirements)

Note: On UNIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status.

Customers should view the [IBM Crypto for C \(ICC\) certificate](#) and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the [NIST CMVP modules in process list](#).

Table 359. Values of constants		
Name	Decimal value	Hexadecimal value
MQSSL_FIPS_NO	0	X'00000000'
MQSSL_FIPS_YES	1	X'00000001'

MQSTAT_* (Stat Options)

Table 360. Values of constants		
Name	Decimal value	Hexadecimal value
MQSTAT_TYPE_ASYNC_ERROR	0	X'00000000'
MQSTAT_TYPE_RECONNECTION	0	X'00000000'
MQSTAT_TYPE_RECONNECTION_ERROR	0	X'00000000'

MQSTS_* (Status reporting structure structure)

Table 361. Structures of constants	
Name	Structure
MQSTS_STRUC_ID	"STAT"
MQSTS_STRUC_ID_ARRAY	'S', 'T', 'A', 'T'

Note: The symbol ◻ represents a single blank character.

Table 362. Values of constants		
Name	Decimal value	Hexadecimal value
MQSTS_VERSION_1	1	X'00000001'
MQSTS_CURRENT_VERSION	1	X'00000001'

MQSUB_* (Durable subscriptions)

Durable subscriptions

Table 363. Values of constants		
Name	Decimal value	Hexadecimal value
MQSUB_DURABLE_AS_PARENT	0	X'00000000'
MQSUB_DURABLE_ALLOWED	1	X'00000001'
MQSUB_DURABLE_INHIBITED	2	X'00000002'

Durable Subscriptions

Table 364. Values of constants		
Name	Decimal value	Hexadecimal value
MQSUB_DURABLE_ALL	-1	X'FFFFFFFF'
MQSUB_DURABLE_YES	1	X'00000001'
MQSUB_DURABLE_NO	2	X'00000002'

MQSUBTYPE_* (Command format Subscription Types)

Table 365. Values of constants		
Name	Decimal value	Hexadecimal value
MQSUBTYPE_API	1	X'00000001'
MQSUBTYPE_ADMIN	2	X'00000002'
MQSUBTYPE_PROXY	3	X'00000003'
MQSUBTYPE_ALL	-1	X'FFFFFFFF'
MQSUBTYPE_USER	-2	X'FFFFFFFE'

MQSUS_* (Command format Suspend Status)

Table 366. Values of constants		
Name	Decimal value	Hexadecimal value
MQSUS_YES	1	X'00000001'
MQSUS_NO	0	X'00000000'

MQSVC_* (Service)

Service Types

Table 367. Values of constants		
Name	Decimal value	Hexadecimal value
MQSVC_TYPE_COMMAND	0	X'00000000'
MQSVC_TYPE_SERVER	1	X'00000001'

Service Controls

Table 368. Values of constants		
Name	Decimal value	Hexadecimal value
MQSVC_CONTROL_Q_MGR	0	X'00000000'
MQSVC_CONTROL_Q_MGR_START	1	X'00000001'
MQSVC_CONTROL_MANUAL	2	X'00000002'

Service Status

Table 369. Values of constants		
Name	Decimal value	Hexadecimal value
MQSVC_STATUS_STOPPED	0	X'00000000'
MQSVC_STATUS_STARTING	1	X'00000001'
MQSVC_STATUS_RUNNING	2	X'00000002'
MQSVC_STATUS_STOPPING	3	X'00000003'
MQSVC_STATUS_RETRYING	4	X'00000004'

MQSYNCPOINT_* (Command format Syncpoint values for Pub/Sub migration)

Table 370. Values of constants		
Name	Decimal value	Hexadecimal value
MQSYNCPOINT_YES	0	X'00000000'
MQSYNCPOINT_IFPER	1	X'00000001'

MQSYSP_* (Command format System Parameter Values)

Table 371. Values of constants		
Name	Decimal value	Hexadecimal value
MQSYSP_NO	0	X'00000000'
MQSYSP_YES	1	X'00000001'
MQSYSP_EXTENDED	2	X'00000002'
MQSYSP_TYPE_INITIAL	10	X'0000000A'
MQSYSP_TYPE_SET	11	X'0000000B'
MQSYSP_TYPE_LOG_COPY	12	X'0000000C'
MQSYSP_TYPE_LOG_STATUS	13	X'0000000D'
MQSYSP_TYPE_ARCHIVE_TAPE	14	X'0000000E'
MQSYSP_ALLOC_BLK	20	X'00000014'
MQSYSP_ALLOC_TRK	21	X'00000015'
MQSYSP_ALLOC_CYL	22	X'00000016'
MQSYSP_STATUS_BUSY	30	X'0000001E'
MQSYSP_STATUS_PREMOUNT	31	X'0000001F'
MQSYSP_STATUS_AVAILABLE	32	X'00000020'
MQSYSP_STATUS_UNKNOWN	33	X'00000021'
MQSYSP_STATUS_ALLOC_ARCHIVE	34	X'00000022'
MQSYSP_STATUS_COPYING_BSDS	35	X'00000023'
MQSYSP_STATUS_COPYING_LOG	36	X'00000024'

MQTA_* (Topic attributes)

Wildcards

Table 372. Values of constants		
Name	Decimal value	Hexadecimal value
MQTA_BLOCK	1	X'00000001'
MQTA_PASSTHRU	2	X'00000002'

Subscriptions Allowed

Table 373. Values of constants		
Name	Decimal value	Hexadecimal value
MQTA_SUB_AS_PARENT	0	X'00000000'
MQTA_SUB_INHIBITED	1	X'00000001'

Table 373. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQTA_SUB_ALLOWED	2	X'00000002'

Proxy Sub Propagation

Table 374. Values of constants		
Name	Decimal value	Hexadecimal value
MQTA_PROXY_SUB_FORCE	1	X'00000001'
MQTA_PROXY_SUB_FIRSTUSE	2	X'00000002'

Publications Allowed

Table 375. Values of constants		
Name	Decimal value	Hexadecimal value
MQTA_PUB_AS_PARENT	0	X'00000000'
MQTA_PUB_INHIBITED	1	X'00000001'
MQTA_PUB_ALLOWED	2	X'00000002'

MQTC_* (Trigger Controls)

Table 376. Values of constants		
Name	Decimal value	Hexadecimal value
MQTC_OFF	0	X'00000000'
MQTC_ON	1	X'00000001'

MQTCPKEEP_* (TCP Keepalive)

Table 377. Values of constants		
Name	Decimal value	Hexadecimal value
MQTCPKEEP_NO	0	X'00000000'
MQTCPKEEP_YES	1	X'00000001'

MQTCPSTACK_* (TCP Stack Types)

Table 378. Values of constants		
Name	Decimal value	Hexadecimal value
MQTCPSTACK_SINGLE	0	X'00000000'
MQTCPSTACK_MULTIPLE	1	X'00000001'

MQTIME_* (Command format Time units)

Table 379. Values of constants		
Name	Decimal value	Hexadecimal value
MQTIME_UNIT_MINS	0	X'00000000'
MQTIME_UNIT_SECS	1	X'00000001'

MQTM_* (Trigger message structure)

Table 380. Structures of constants	
Name	Structure
MQTM_STRUC_ID	"TM↵"
MQTM_STRUC_ID_ARRAY	'T','M','↵','↵'

Note: The symbol ↵ represents a single blank character.

Table 381. Values of constants		
Name	Decimal value	Hexadecimal value
MQTM_VERSION_1	1	X'00000001'
MQTM_CURRENT_VERSION	1	X'00000001'

MQTMC_* (Trigger message character format structure)

Table 382. Structures of constants	
Name	Structure
MQTMC_STRUC_ID	"TMC↵"
MQTMC_STRUC_ID_ARRAY	'T','M','C','↵'
MQTMC_VERSION_1	"↵↵1"
MQTMC_VERSION_2	"↵↵2"
MQTMC_CURRENT_VERSION	"↵↵2"
MQTMC_VERSION_1_ARRAY	'↵','↵','↵','1'
MQTMC_VERSION_2_ARRAY	'↵','↵','↵','2'
MQTMC_CURRENT_VERSION_ARRAY	'↵','↵','↵','2'

MQTOPT_* (Topic Type)

Table 383. Values of constants		
Name	Decimal value	Hexadecimal value
MQTOPT_LOCAL	0	X'00000000'
MQTOPT_CLUSTER	1	X'00000001'
MQTOPT_ALL	2	X'00000002'

MQTRAXSTR_* (Channel Initiator Trace Autostart)

Table 384. Values of constants		
Name	Decimal value	Hexadecimal value
MQTRAXSTR_NO	0	X'00000000'
MQTRAXSTR_YES	1	X'00000001'

MQTScope_* (Subscription Scope)

Table 385. Values of constants		
Name	Decimal value	Hexadecimal value
MQTScope_QMGR	1	X'00000001'
MQTScope_ALL	2	X'00000002'

MQTT_* (Trigger Types)

Table 386. Values of constants		
Name	Decimal value	Hexadecimal value
MQTT_NONE	0	X'00000000'
MQTT_FIRST	1	X'00000001'
MQTT EVERY	2	X'00000002'
MQTT_DEPTH	3	X'00000003'

MQTYPE_* (Property data types)

Table 387. Values of constants		
Name	Decimal value	Hexadecimal value
MQTYPE_AS_SET	0	X'00000000'
MQTYPE_NULL	2	X'00000002'
MQTYPE_BOOLEAN	4	X'00000004'
MQTYPE_BYTE_STRING	8	X'00000008'
MQTYPE_INT8	16	X'00000010'
MQTYPE_INT16	32	X'00000020'
MQTYPE_INT32	64	X'00000040'
MQTYPE_LONG	64	X'00000040'
MQTYPE_INT64	128	X'00000080'
MQTYPE_FLOAT32	256	X'00000100'
MQTYPE_FLOAT64	512	X'00000200'
MQTYPE_STRING	1024	X'00000400'

MQUA_* (Publish/Subscribe User Attribute Selectors)

Table 388. Values of constants		
Name	Decimal value	Hexadecimal value
MQUA_FIRST	65536	X'00010000'
MQUA_LAST	999999999	X'3B9AC9FF'

MQUIDSUPP_* (Command format User ID Support)

Table 389. Values of constants		
Name	Decimal value	Hexadecimal value
MQUIDSUPP_NO	0	X'00000000'
MQUIDSUPP_YES	1	X'00000001'

MQUNDELIVERED_* (Command format Undelivered values for Pub/Sub migration)

Table 390. Values of constants		
Name	Decimal value	Hexadecimal value
MQUNDELIVERED_NORMAL	0	X'00000000'

Table 390. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQUNDELIVERED_SAFE	1	X'00000001'
MQUNDELIVERED_DISCARD	2	X'00000002'
MQUNDELIVERED_KEEP	3	X'00000003'

MQUOWST_* (Command format UOW States)

Table 391. Values of constants		
Name	Decimal value	Hexadecimal value
MQUOWST_NONE	0	X'00000000'
MQUOWST_ACTIVE	1	X'00000001'
MQUOWST_PREPARED	2	X'00000002'
MQUOWST_UNRESOLVED	3	X'00000003'

MQUOWT_* (Command format UOW Types)

Table 392. Values of constants		
Name	Decimal value	Hexadecimal value
MQUOWT_Q_MGR	0	X'00000000'
MQUOWT_CICS	1	X'00000001'
MQUOWT_RRS	2	X'00000002'
MQUOWT_IMS	3	X'00000003'
MQUOWT_XA	4	X'00000004'

MQUS_* (Queue Usages)

Table 393. Values of constants		
Name	Decimal value	Hexadecimal value
MQUS_NORMAL	0	X'00000000'
MQUS_TRANSMISSION	1	X'00000001'

MQUSAGE_* (Command format Page Set Usage Values and Data Set Usage Values)

Command format Page Set Usage Values

Table 394. Values of constants		
Name	Decimal value	Hexadecimal value
MQUSAGE_PS_AVAILABLE	0	X'00000000'
MQUSAGE_PS_DEFINED	1	X'00000001'
MQUSAGE_PS_OFFLINE	2	X'00000002'
MQUSAGE_PS_NOT_DEFINED	3	X'00000003'
MQUSAGE_PS_SUSPENDED	4	X'00000004'
MQUSAGE_EXPAND_USER	1	X'00000001'
MQUSAGE_EXPAND_SYSTEM	2	X'00000002'

Table 394. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQUSAGE_EXPAND_NONE	3	X'00000003'

Command format Data Set Usage Values

Table 395. Values of constants		
Name	Decimal value	Hexadecimal value
MQUSAGE_DS_OLDEST_ACTIVE_UOW	10	X'0000000A'
MQUSAGE_DS_OLDEST_PS_RECOVERY	11	X'0000000B'
MQUSAGE_DS_OLDEST_CF_RECOVERY	12	X'0000000C'

MQVL_* (Value Length)

Table 396. Values of constants		
Name	Decimal value	Hexadecimal value
MQVL_NULL_TERMINATED	-1	X'FFFFFFFF'
MQVL_EMPTY_STRING	0	X'00000000'

MQVU_* (Variable User ID)

Table 397. Values of constants		
Name	Decimal value	Hexadecimal value
MQVU_FIXED_USER	1	X'00000001'
MQVU_ANY_USER	2	X'00000002'

MQWDR_* (Cluster workload exit destination record structure)

Table 398. Structures of constants	
Name	Structure
MQWDR_STRUC_ID	"WDR¬"
MQWDR_STRUC_ID_ARRAY	'W','D','R','¬'

Note: The symbol ¬ represents a single blank character.

Table 399. Values of constants		
Name	Decimal value	Hexadecimal value
MQWDR_VERSION_1	1	X'00000001'
MQWDR_VERSION_2	2	X'00000002'
MQWDR_CURRENT_VERSION	2	X'00000002'
MQWDR_LENGTH_1	124	X'0000007C'
MQWDR_LENGTH_2	136	X'00000088'
MQWDR_CURRENT_LENGTH	136	X'00000088'

MQWI_* (Wait Interval)

Table 400. Values of constants		
Name	Decimal value	Hexadecimal value
MQWI_UNLIMITED	-1	X'FFFFFFFF'

MQWIH_* (Workload information header structure and Flags)

Workload information header structure

Table 401. Structures of constants	
Name	Structure
MQWIH_STRUC_ID	"WIH"
MQWIH_STRUC_ID_ARRAY	'W','I','H',' '

Note: The symbol `␣` represents a single blank character.

Table 402. Values of constants		
Name	Decimal value	Hexadecimal value
MQWIH_VERSION_1	1	X'00000001'
MQWIH_CURRENT_VERSION	1	X'00000001'
MQWIH_LENGTH_1	120	X'00000078'
MQWIH_CURRENT_LENGTH	120	X'00000078'

Workload information header Flags

Table 403. Values of constants		
Name	Decimal value	Hexadecimal value
MQWIH_NONE	0	X'00000000'

MQWQR_* (Cluster workload exit queue record structure)

Table 404. Structures of constants	
Name	Structure
MQWQR_STRUC_ID	"WQR"
MQWQR_STRUC_ID_ARRAY	'W','Q','R',' '

Note: The symbol `␣` represents a single blank character.

Table 405. Values of constants		
Name	Decimal value	Hexadecimal value
MQWQR_VERSION_1	1	X'00000001'
MQWQR_VERSION_2	2	X'00000002'
MQWQR_VERSION_3	3	X'00000003'
MQWQR_CURRENT_VERSION	3	X'00000003'
MQWQR_LENGTH_1	200	X'000000C8'
MQWQR_LENGTH_2	208	X'000000D0'
MQWQR_LENGTH_3	212	X'000000D4'

Table 405. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQWQR_CURRENT_LENGTH	212	X'000000D4'

MQWS_* (Wildcard Schema)

Table 406. Values of constants		
Name	Decimal value	Hexadecimal value
MQWS_DEFAULT	0	X'00000000'
MQWS_CHAR	1	X'00000001'
MQWS_TOPIC	2	X'00000002'

MQWXP_* (Cluster workload exit parameter structure)

MQWXP_* (Cluster workload exit parameter structure)

Table 407. Structures of constants	
Name	Structure
MQWXP_STRUC_ID	"WXP¬"
MQWXP_STRUC_ID_ARRAY	'W', 'X', 'P', '¬'

Note: The symbol ¬ represents a single blank character.

Table 408. Values of constants		
Name	Decimal value	Hexadecimal value
MQWXP_VERSION_1	1	X'00000001'
MQWXP_VERSION_2	2	X'00000002'
MQWXP_VERSION_3	3	X'00000003'
MQWXP_VERSION_4	4	X'00000004'
MQWXP_CURRENT_VERSION	4	X'00000004'

MQWXP_* (Cluster Workload Flags)

Table 409. Values of constants		
Name	Decimal value	Hexadecimal value
MQWXP_PUT_BY_CLUSTER_CHL	2	X'00000002'

Related reference

“Fields in MQWXP - Cluster workload exit parameter structure” on page 1549
Description of the fields in the MQWXP - Cluster workload exit parameter structure

MQXACT_* (API Caller Types)

Table 410. Values of constants		
Name	Decimal value	Hexadecimal value
MQXACT_EXTERNAL	1	X'00000001'
MQXACT_INTERNAL	2	X'00000002'

MQXC_* (Exit Commands)

Table 411. Values of constants		
Name	Decimal value	Hexadecimal value
MQXC_MQOPEN	1	X'00000001'
MQXC_MQCLOSE	2	X'00000002'
MQXC_MQGET	3	X'00000003'
MQXC_MQPUT	4	X'00000004'
MQXC_MQPUT1	5	X'00000005'
MQXC_MQINQ	6	X'00000006'
MQXC_MQSET	8	X'00000008'
MQXC_MQBACK	9	X'00000009'
MQXC_MQCMIT	10	X'0000000A'

MQXCC_* (Exit Responses)

Table 412. Values of constants		
Name	Decimal value	Hexadecimal value
MQXCC_OK	0	X'00000000'
MQXCC_SUPPRESS_FUNCTION	-1	X'FFFFFFFF'
MQXCC_SKIP_FUNCTION	-2	X'FFFFFFFE'
MQXCC_SEND_AND_REQUEST_SEC_MSG	-3	X'FFFFFFFD'
MQXCC_SEND_SEC_MSG	-4	X'FFFFFFFC'
MQXCC_SUPPRESS_EXIT	-5	X'FFFFFFFB'
MQXCC_CLOSE_CHANNEL	-6	X'FFFFFFFA'
MQXCC_REQUEST_ACK	-7	X'FFFFFFF9'
MQXCC_FAILED	-8	X'FFFFFFF8'

MQXDR_* (Exit Response)

Table 413. Values of constants		
Name	Decimal value	Hexadecimal value
MQXDR_OK	0	X'00000000'
MQXDR_CONVERSION_FAILED	1	X'00000001'

MQXE_* (Environments)

Table 414. Values of constants		
Name	Decimal value	Hexadecimal value
MQXE_OTHER	0	X'00000000'
MQXE_MCA	1	X'00000001'
MQXE_MCA_SVRCONN	2	X'00000002'
MQXE_COMMAND_SERVER	3	X'00000003'
MQXE_MQSC	4	X'00000004'

MQXEPO_* (Register Entry Point Options structure and Exit Options)

Register Entry Point Options structure

Table 415. Structures of constants	
Name	Structure
MQXEPO_STRUC_ID	"XEPO"
MQXEPO_STRUC_ID_ARRAY	'X', 'E', 'P', 'O'

Note: The symbol ␣ represents a single blank character.

Table 416. Values of constants		
Name	Decimal value	Hexadecimal value
MQXEPO_VERSION_1	1	X'00000001'
MQXEPO_CURRENT_VERSION	1	X'00000001'

Exit Options

Table 417. Values of constants		
Name	Decimal value	Hexadecimal value
MQXEPO_NONE	0	X'00000000'

MQXF_* (API Function Identifiers)

Table 418. Values of constants		
Name	Decimal value	Hexadecimal value
MQXF_INIT	1	X'00000001'
MQXF_TERM	2	X'00000002'
MQXF_CONN	3	X'00000003'
MQXF_CONNX	4	X'00000004'
MQXF_DISC	5	X'00000005'
MQXF_OPEN	6	X'00000006'
MQXF_CLOSE	7	X'00000007'
MQXF_PUT1	8	X'00000008'
MQXF_PUT	9	X'00000009'
MQXF_GET	10	X'0000000A'
MQXF_DATA_CONV_ON_GET	11	X'0000000B'
MQXF_INQ	12	X'0000000C'
MQXF_SET	13	X'0000000D'
MQXF_BEGIN	14	X'0000000E'
MQXF_CMIT	15	X'0000000F'
MQXF_BACK	16	X'00000010'
MQXF_STAT	18	X'00000012'
MQXF_CB	19	X'00000013'
MQXF_CTL	20	X'00000014'
MQXF_CALLBACK	21	X'00000015'

Table 418. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQXF_SUB	22	X'00000016'
MQXF_SUBRQ	23	X'00000017'
MQXF_XACLOSE	24	X'00000018'
MQXF_XACOMMIT	25	X'00000019'
MQXF_XACOMplete	26	X'0000001A'
MQXF_XAEND	27	X'0000001B'
MQXF_XAFORGET	28	X'0000001C'
MQXF_XAOPEN	29	X'0000001D'
MQXF_XAPREPARE	30	X'0000001E'
MQXF_XARECOVER	31	X'0000001F'
MQXF_XAROLLBACK	32	X'00000020'
MQXF_XASTART	33	X'00000021'
MQXF_AXREG	34	X'00000022'
MQXF_AXUNREG	35	X'00000023'

MQXP_* (API crossing exit parameter structure)

Table 419. Structures of constants	
Name	Structure
MQXP_STRUC_ID	"XP¬"
MQXP_STRUC_ID_ARRAY	'X','P','¬','¬'

Note: The symbol ¬ represents a single blank character.

Table 420. Values of constants		
Name	Decimal value	Hexadecimal value
MQXP_VERSION_1	1	X'00000001'

MQXPDA_* (Problem Determination Area)

Table 421. Constant names and values	
Name	Value
MQXPDA_NONE	X'00...00' (48 nulls)
MQXPDA_NONE_ARRAY	'\0','\0',... (48 nulls)

MQXPT_* (Transport Types)

Table 422. Values of constants		
Name	Decimal value	Hexadecimal value
MQXPT_ALL	-1	X'FFFFFFFF'
MQXPT_LOCAL	0	X'00000000'
MQXPT_LU62	1	X'00000001'
MQXPT_TCP	2	X'00000002'
MQXPT_NETBIOS	3	X'00000003'

Table 422. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQXPT_SPX	4	X'00000004'
MQXPT_DECNET	5	X'00000005'
MQXPT_UDP	6	X'00000006'

MQXQH_* (Transmission queue header structure)

Table 423. Structures of constants	
Name	Structure
MQXQH_STRUC_ID	"XQH¬"
MQXQH_STRUC_ID_ARRAY	'X','Q','H','¬'

Note: The symbol ¬ represents a single blank character.

Table 424. Values of constants		
Name	Decimal value	Hexadecimal value
MQXQH_VERSION_1	1	X'00000001'
MQXQH_CURRENT_VERSION	1	X'00000001'

MQXR_* (Exit Reasons)

Table 425. Values of constants		
Name	Decimal value	Hexadecimal value
MQXR_BEFORE	1	X'00000001'
MQXR_AFTER	2	X'00000002'
MQXR_CONNECTION	3	X'00000003'
MQXR_INIT	11	X'0000000B'
MQXR_TERM	12	X'0000000C'
MQXR_MSG	13	X'0000000D'
MQXR_XMIT	14	X'0000000E'
MQXR_SEC_MSG	15	X'0000000F'
MQXR_INIT_SEC	16	X'00000010'
MQXR_RETRY	17	X'00000011'
MQXR_AUTO_CLUSSDR	18	X'00000012'
MQXR_AUTO_RECEIVER	19	X'00000013'
MQXR_CLWL_OPEN	20	X'00000014'
MQXR_CLWL_PUT	21	X'00000015'
MQXR_CLWL_MOVE	22	X'00000016'
MQXR_CLWL_REPOS	23	X'00000017'
MQXR_CLWL_REPOS_MOVE	24	X'00000018'
MQXR_END_BATCH	25	X'00000019'
MQXR_ACK_RECEIVED	26	X'0000001A'
MQXR_AUTO_SVRCONN	27	X'0000001B'
MQXR_AUTO_CLUSRCVR	28	X'0000001C'

Table 425. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQXR_SEC_PARMS	29	X'0000001D'

MQXR2_* (Exit Response 2)

Table 426. Values of constants		
Name	Decimal value	Hexadecimal value
MQXR2_PUT_WITH_DEF_ACTION	0	X'00000000'
MQXR2_PUT_WITH_DEF_USERID	1	X'00000001'
MQXR2_PUT_WITH_MSG_USERID	2	X'00000002'
MQXR2_USE_AGENT_BUFFER	0	X'00000000'
MQXR2_USE_EXIT_BUFFER	4	X'00000004'
MQXR2_DEFAULT_CONTINUATION	0	X'00000000'
MQXR2_CONTINUE_CHAIN	8	X'00000008'
MQXR2_SUPPRESS_CHAIN	16	X'00000010'
MQXR2_STATIC_CACHE	0	X'00000000'
MQXR2_DYNAMIC_CACHE	32	X'00000020'

MQXT_* (Exit Identifiers)

Table 427. Values of constants		
Name	Decimal value	Hexadecimal value
MQXT_API_CROSSING_EXIT	1	X'00000001'
MQXT_API_EXIT	2	X'00000002'
MQXT_CHANNEL_SEC_EXIT	11	X'0000000B'
MQXT_CHANNEL_MSG_EXIT	12	X'0000000C'
MQXT_CHANNEL_SEND_EXIT	13	X'0000000D'
MQXT_CHANNEL_RCV_EXIT	14	X'0000000E'
MQXT_CHANNEL_MSG_RETRY_EXIT	15	X'0000000F'
MQXT_CHANNEL_AUTO_DEF_EXIT	16	X'00000010'
MQXT_CLUSTER_WORKLOAD_EXIT	20	X'00000014'
MQXT_PUBSUB_ROUTING_EXIT	21	X'00000015'

MQXUA_* (Exit User Area Value)

Table 428. Constant names and values	
Name	Value
MQXUA_NONE	X'00...00' (16 nulls)
MQXUA_NONE_ARRAY	'\0','\0',... (16 nulls)

MQXWD_* (Exit wait descriptor structure)

Table 429. Structures of constants	
Name	Structure
MQXWD_STRUC_ID	"XWD-"

MQZAO_* (Installable Services Authorizations)

Table 436. Values of constants		
Name	Decimal value	Hexadecimal value
MQZAO_CONNECT	1	X'00000001'
MQZAO_BROWSE	2	X'00000002'
MQZAO_INPUT	4	X'00000004'
MQZAO_OUTPUT	8	X'00000008'
MQZAO_INQUIRE	16	X'00000010'
MQZAO_SET	32	X'00000020'
MQZAO_PASS_IDENTITY_CONTEXT	64	X'00000040'
MQZAO_PASS_ALL_CONTEXT	128	X'00000080'
MQZAO_SET_IDENTITY_CONTEXT	256	X'00000100'
MQZAO_SET_ALL_CONTEXT	512	X'00000200'
MQZAO_ALTERNATE_USER_AUTHORITY	1024	X'00000400'
MQZAO_PUBLISH	2048	X'00000800'
MQZAO_SUBSCRIBE	4096	X'00001000'
MQZAO_RESUME	8192	X'00002000'
MQZAO_ALL_MQI	16383	X'00003FFF'
MQZAO_CREATE	65536	X'00010000'
MQZAO_DELETE	131072	X'00020000'
MQZAO_DISPLAY	262144	X'00040000'
MQZAO_CHANGE	524288	X'00080000'
MQZAO_CLEAR	1048576	X'00100000'
MQZAO_CONTROL	2097152	X'00200000'
MQZAO_CONTROL_EXTENDED	4194304	X'00400000'
MQZAO_AUTHORIZE	8388608	X'00800000'
MQZAO_ALL_ADMIN	16646144	X'00FE0000'
MQZAO_ALL	16662527	X'00FE3FFF'
MQZAO_REMOVE	16777216	X'01000000'
MQZAO_NONE	0	X'00000000'

MQZAS_* (Installable Services Service Interface Version)

Table 437. Values of constants		
Name	Decimal value	Hexadecimal value
MQZAS_VERSION_1	1	X'00000001'
MQZAS_VERSION_2	2	X'00000002'
MQZAS_VERSION_3	3	X'00000003'
MQZAS_VERSION_4	4	X'00000004'
MQZAS_VERSION_5	5	X'00000005'
MQZAS_VERSION_6	6	X'00000006'

MQZAT_* (Authentication Types)

Table 438. Values of constants		
Name	Decimal value	Hexadecimal value
MQZAT_INITIAL_CONTEXT	0	X'00000000'
MQZAT_CHANGE_CONTEXT	1	X'00000001'

MQZCI_* (Installable Services Continuation Indicator)

Table 439. Values of constants		
Name	Decimal value	Hexadecimal value
MQZCI_DEFAULT	0	X'00000000'
MQZCI_CONTINUE	0	X'00000000'
MQZCI_STOP	1	X'00000001'

MQZED_* (Entity data structure)

Table 440. Structures of constants	
Name	Structure
MQZED_STRUC_ID	"ZED¬"
MQZED_STRUC_ID_ARRAY	'Z','E','D','¬'

Note: The symbol ¬ represents a single blank character.

Table 441. Values of constants		
Name	Decimal value	Hexadecimal value
MQZED_VERSION_1	1	X'00000001'
MQZED_VERSION_2	2	X'00000002'
MQZED_CURRENT_VERSION	2	X'00000002'

MQZFP_* (Free parameters structure)

Table 442. Structures of constants	
Name	Structure
MQZFP_STRUC_ID	"ZFP¬"
MQZFP_STRUC_ID_ARRAY	'Z','F','P','¬'

Note: The symbol ¬ represents a single blank character.

Table 443. Values of constants		
Name	Decimal value	Hexadecimal value
MQZFP_VERSION_1	1	X'00000001'
MQZFP_CURRENT_VERSION	1	X'00000001'

MQZIC_* (Identity context structure)

Table 444. Structures of constants	
Name	Structure
MQZIC_STRUC_ID	"ZIC¬"

Table 444. Structures of constants (continued)

Name	Structure
MQZIC_STRUC_ID_ARRAY	'Z', 'I', 'C', 'H'

Note: The symbol \neg represents a single blank character.

Table 445. Values of constants

Name	Decimal value	Hexadecimal value
MQZIC_VERSION_1	1	X'00000001'
MQZIC_CURRENT_VERSION	1	X'00000001'

MQZID_* (Function ids for services)

Function ids common to all services

Table 446. Values of constants

Name	Decimal value	Hexadecimal value
MQZID_INIT	0	X'00000000'
MQZID_TERM	1	X'00000001'

Function ids for Authority service

Table 447. Values of constants

Name	Decimal value	Hexadecimal value
MQZID_INIT_AUTHORITY	0	X'00000000'
MQZID_TERM_AUTHORITY	1	X'00000001'
MQZID_CHECK_AUTHORITY	2	X'00000002'
MQZID_COPY_ALL_AUTHORITY	3	X'00000003'
MQZID_DELETE_AUTHORITY	4	X'00000004'
MQZID_SET_AUTHORITY	5	X'00000005'
MQZID_GET_AUTHORITY	6	X'00000006'
MQZID_GET_EXPLICIT_AUTHORITY	7	X'00000007'
MQZID_REFRESH_CACHE	8	X'00000008'
MQZID_ENUMERATE_AUTHORITY_DATA	9	X'00000009'
MQZID_AUTHENTICATE_USER	10	X'0000000A'
MQZID_FREE_USER	11	X'0000000B'
MQZID_INQUIRE	12	X'0000000C'
MQZID_CHECK_PRIVILEGED	13	X'0000000D'

Function ids for Name service

Table 448. Values of constants

Name	Decimal value	Hexadecimal value
MQZID_INIT_NAME	0	X '00000000 '
MQZID_TERM_NAME	1	X '00000001 '
MQZID_LOOKUP_NAME	2	X '00000002 '

Table 448. Values of constants (continued)		
Name	Decimal value	Hexadecimal value
MQZID_INSERT_NAME	3	X'00000003'
MQZID_DELETE_NAME	4	X'00000004'

Function ids for Userid service

Table 449. Values of constants		
Name	Decimal value	Hexadecimal value
MQZID_INIT_USERID	0	X'00000000'
MQZID_TERM_USERID	1	X'00000001'
MQZID_FIND_USERID	2	X'00000002'

MQZIO_* (Installable Services Initialization Options)

Table 450. Values of constants		
Name	Decimal value	Hexadecimal value
MQZIO_PRIMARY	0	X'00000000'
MQZIO_SECONDARY	1	X'00000001'

MQZNS_* (Name Service Interface Version)

Table 451. Values of constants		
Name	Decimal value	Hexadecimal value
MQZNS_VERSION_1	1	X'00000001'

MQZSE_* (Installable Services Start-Enumeration Indicator)

Table 452. Values of constants		
Name	Decimal value	Hexadecimal value
MQZSE_START	1	X'00000001'
MQZSE_CONTINUE	0	X'00000000'

MQZSL_* (Installable Services Selector Indicator)

Table 453. Values of constants		
Name	Decimal value	Hexadecimal value
MQZSL_NOT_RETURNED	0	X'00000000'
MQZSL_RETURNED	1	X'00000001'

MQZTO_* (Installable Services Termination Options)

Table 454. Values of constants		
Name	Decimal value	Hexadecimal value
MQZTO_PRIMARY	0	X'00000000'
MQZTO_SECONDARY	1	X'00000001'

MQZUS_* (Userid Service Interface Version)

Table 455. Values of constants		
Name	Decimal value	Hexadecimal value
MQZUS_VERSION_1	1	X'00000001'

Data types used in the MQI

Information on the data types that can be used in the Message Queue Interface (MQI). Descriptions, fields, and language declarations for relevant languages with each data type.

Introducing data types used in the MQI

This section introduces the data types used in the MQI, and gives you some guidance on using them in the supported programming languages.

Elementary data types

This section contains information about data types used in the MQI (or in exit functions). These are described in detail, followed by examples showing how to declare the elementary data types in the supported programming languages in the following topics.

The data types used in the MQI (or in exit functions) are either:

- Elementary data types, or
- Aggregates of elementary data types (arrays or structures)

The following elementary data types are used in the MQI (or in exit functions):

Elementary data type name	Data type	Description
MQBOOL	Boolean	The MQBOOL data type represents a boolean value. The value 0 represents false. Any other value represents true. An MQBOOL must be aligned as for the MQLONG data type.

Elementary data type name	Data type	Description
MQBYTE	Byte	<p>The MQBYTE data type represents a single byte of data. No particular interpretation is placed on the byte; it is treated as a string of bits, and not as a binary number or character. No special alignment is required.</p> <p>When MQBYTE data is sent between queue managers that use different character sets or encodings, the MQBYTE data is <i>not</i> converted in any way. The <i>MsgId</i> and <i>CorrelId</i> fields in the MQMD structure are like this.</p> <p>An array of MQBYTE is sometimes used to represent an area of main storage that is not known to the queue manager. For example, the area might contain application message data or a structure. The boundary alignment of this area must be compatible with the nature of the data contained within it.</p> <p>In the C programming language, any data type can be used for function parameters that are shown as arrays of MQBYTE. This is because such parameters are always passed by address, and in C the function parameter is declared as a pointer-to-void.</p>

Elementary data type name	Data type	Description
MQBYTEn	String of <i>n</i> bytes	<p>Each MQBYTEn data type represents a string of <i>n</i> bytes, where <i>n</i> can take any of the following values: 8, 16, 24, 32, 40, or 128. Each byte is described by the MQBYTE data type. No special alignment is required.</p> <p>If the data in the byte string is shorter than the defined length of the string, the data must be padded with nulls to fill the string.</p> <p>When the queue manager returns byte strings to the application (for example, on the MQGET call), the queue manager pads with nulls to the defined length of the string.</p> <p>Named constants are available to define the lengths of byte string fields. These are listed in “Constants” on page 59</p>
MQCHAR	Character	<p>The MQCHAR data type represents a single-byte character, or one byte of a double-byte or multi-byte character. No special alignment is required.</p> <p>When MQCHAR data is sent between queue managers that use different character sets or encodings, the MQCHAR data usually requires conversion in order for the data to be interpreted correctly. The queue manager does this automatically for MQCHAR data in the MQMD structure. Conversion of MQCHAR data in the application message data is controlled by the MQGMO_CONVERT option specified on the MQGET call; see the description of this option in “MQGMO - Get-message options” on page 350 for further details.</p>

Elementary data type name	Data type	Description
MQCHARn	String of <i>n</i> characters	<p>Each MQCHARn data type represents a string of <i>n</i> characters, where <i>n</i> can take any of the following values: 4, 8, 12, 20, 28, 32, 48, 64, 128, or 256. Each character is described by the MQCHAR data type. No special alignment is required.</p> <p>If the data in the string is shorter than the defined length of the string, the data must be padded with blanks to fill the string. In some cases a null character can be used to end the string prematurely, instead of padding with blanks; the null character and characters following it are treated as blanks, up to the defined length of the string. The places where a null can be used are identified in the call and data type descriptions.</p> <p>When the queue manager returns character strings to the application (for example, on the MQGET call), the queue manager always pads with blanks to the defined length of the string; the queue manager does not use the null character to delimit the string.</p> <p>Named constants are available that define the lengths of character string fields and are listed in “Constants” on page 59.</p>
MQFLOAT32	32-bit floating point number	<p>The MQFLOAT32 data type is a 32-bit floating-point number represented using the standard IEEE floating-point format. An MQFLOAT32 must be aligned on a 4-byte boundary.</p> <p>The use of MQFLOAT32 in C on z/OS requires the use of the FLOAT(IEEE) compiler flag.</p> <p>The use of MQFLOAT32 in COBOL is limited to compilers that support floating-point numbers in IEEE format. This might require the use of the FLOAT(NATIVE) compiler flag.</p>

Elementary data type name	Data type	Description
MQFLOAT64	64-bit floating point number	<p>The MQFLOAT64 data type is a 64-bit floating-point number represented using the standard IEEE floating-point format. An MQFLOAT64 must be aligned on an 8-byte boundary.</p> <p>The use of MQFLOAT64 in C on z/OS requires the use of the FLOAT(IEEE) compiler flag.</p> <p>The use of MQFLOAT64 in COBOL is limited to compilers that support floating-point numbers in IEEE format. This might require the use of the FLOAT(NATIVE) compiler flag.</p>
MQHCONFIG	Configuration handle	<p>The MQHCONFIG data type represents a configuration handle, that is, the component that is being configured for a particular installable service. A configuration handle must be aligned on its natural boundary.</p> <p>Applications must not rely on the format of the data stored inside this handle. If valid, its value is intended to be usable in further MQI calls, but is not intended to have any meaning besides that purpose.</p>
MQHCONN	Connection handle	<p>The MQHCONN data type represents a connection handle, that is, the connection to a particular queue manager. A connection handle must be aligned on a 4-byte boundary.</p> <p>Applications must not rely on the format of the data stored inside this handle. If valid, its value is intended to be usable in further MQI calls, but is not intended to have any meaning besides that purpose.</p>

Elementary data type name	Data type	Description
MQHMSG	Message handle	<p>The MQHMSG data type represents a message handle that gives access to a message. A message handle must be aligned on an 8-byte boundary.</p> <p>Applications must not rely on the format of the data stored inside this handle. If valid, its value is intended to be usable in further MQI calls, but is not intended to have any meaning besides that purpose.</p>
MQHOBJ	Object handle	<p>The MQHOBJ data type represents an object handle that gives access to an object. An object handle must be aligned on a 4-byte boundary.</p> <p>Applications must not rely on the format of the data stored inside this handle. If valid, its value is intended to be usable in further MQI calls, but is not intended to have any meaning besides that purpose.</p>
MQINT8	8-bit signed integer	The MQINT8 data type is an 8-bit signed integer that can take any value in the range -128 to +127, unless otherwise restricted by the context.
MQINT16	16-bit signed integer	The MQINT16 data type is a 16-bit signed integer that can take any value in the range -32 768 to +32 767, unless otherwise restricted by the context. An MQINT16 must be aligned on a 2-byte boundary.
MQINT32	32-bit signed integer	<p>The MQINT32 data type is a 32-bit signed binary integer that can take any value in the range -2 147 483 648 through +2 147 483 647, unless otherwise restricted by the context.</p> <p>See the definition of MQLONG.</p>

Elementary data type name	Data type	Description
MQINT64	64-bit signed integer	<p>The MQINT64 data type is a 64-bit signed integer that can take any value in the range -9 223 372 036 854 775 808 through +9 223 372 036 854 775 807, unless otherwise restricted by the context.</p> <p>For COBOL, the valid range is limited to -999 999 999 999 999 999 through +999 999 999 999 999 999. A 64-bit integer must be aligned on an 8-byte boundary.</p>
MQLONG	32-bit signed integer	<p>The MQLONG data type is a 32-bit signed binary integer that can take any value in the range -2 147 483 648 through +2 147 483 647, unless otherwise restricted by the context.</p> <p>For COBOL, the valid range is limited to -999 999 999 through +999 999 999. An MQLONG must be aligned on a 4-byte boundary.</p>
MQPID	Process identifier	<p>The IBM MQ process identifier.</p> <p>This is the same identifier used in MQ trace and FFST™ dumps, but might be different from the operating system process identifier.</p>
MQPTR	Pointer	<p>The MQPTR data type is the address of data of any type. A pointer must be aligned on its natural boundary; this is a 16-byte boundary on IBM i, and an 8-byte boundary on other platforms.</p> <p>Some programming languages support typed pointers; the MQI also uses these in a few cases (for example, PMQCHAR and PMQLONG in the C programming language).</p>
MQTID	Thread identifier	<p>The IBM MQ thread identifier.</p> <p>This is the same identifier used in MQ trace and FFST™ dumps, but might be different from the operating system thread identifier.</p>

Elementary data type name	Data type	Description
MQUINT8	8-bit unsigned integer	The MQUINT8 data type is an 8-bit unsigned integer that can take any value in the range 0 to +255, unless otherwise restricted by the context.
MQUINT16	16-bit unsigned integer	The MQUINT16 data type is a 16-bit unsigned integer that can take any value in the range 0 through +65 535, unless otherwise restricted by the context. An MQUINT16 must be aligned on a 2-byte boundary.
MQUINT32	32-bit unsigned integer	The MQUINT32 data type is a 32-bit unsigned binary integer. See the definition of MQULONG .
MQUINT64	64-bit unsigned integer	The MQUINT64 data type is a 64-bit unsigned integer that can take any value in the range 0 through +18 446 744 073 709 551 615, unless otherwise restricted by the context. For COBOL, the valid range is limited to 0 through +999 999 999 999 999 999 999. A 64-bit integer must be aligned on an 8-byte boundary.
MQULONG	32-bit unsigned integer	The MQULONG data type is a 32-bit unsigned binary integer that can take any value in the range 0 through +4 294 967 294, unless otherwise restricted by the context. For COBOL, the valid range is limited to 0 through +999 999 999. An MQULONG must be aligned on a 4-byte boundary.
PMQACH	Pointer	Pointer to a data structure of type MQACH
PMQAIR	Pointer	Pointer to a data structure of type MQAIR
PMQAXC	Pointer	Pointer to a data structure of type MQAXC
PMQAXP	Pointer	Pointer to a data structure of type MQAXP
PMQBMHO	Pointer	Pointer to a data structure of type MQBMHO

Elementary data type name	Data type	Description
PMQBO	Pointer	Pointer to a data structure of type MQBO
PMQBOOL	Pointer	Pointer to data of type MQBOOL
PMQBYTE	Pointer	Pointer to data of type MQBYTE
PMQBYTEn	Pointer	Pointer to data of type MQBYTEn, where n can be 8, 16, 24, 32, 40, 128
PMQCBC	Pointer	Pointer to a data structure of type MQCBC
PMQCBD	Pointer	Pointer to a data structure of type MQCBD
PMQCHAR	Pointer	Pointer to data of type MQCHAR
PMQCHARN	Pointer	Pointer to a data type of MQCHARN, where n can be 4, 8, 12, 20, 28, 32, 48, 64, 128, 256, 264
PMQCHARV	Pointer	Pointer to a data structure of type MQCHARV
PMQCIH	Pointer	Pointer to a data structure of type MQCIH
PMQCMHO	Pointer	Pointer to a data structure of type MQCMHO
PMQCNO	Pointer	Pointer to a data structure of type MQCNO
PMQCSP	Pointer	Pointer to a data structure of type MQCSP
PMQCTLO	Pointer	Pointer to a data structure of type MQCTLO
PMQDH	Pointer	Pointer to a data structure of type MQDH
PMQDHO	Pointer	Pointer to a data structure of type MQDHO
PMQDLH	Pointer	Pointer to a data structure of type MQDLH
PMQDMHO	Pointer	Pointer to a data structure of type MQDMHO
PMQDMPO	Pointer	Pointer to a data structure of type MQDMPO
PMQEPH	Pointer	Pointer to a data structure of type MQEPH
PMQFLOAT32	Pointer	Pointer to a data structure of type MQFLOAT32

Elementary data type name	Data type	Description
PMQFLOAT64	Pointer	Pointer to a data structure of type MQFLOAT64
PMQFUNC	Pointer	Pointer to a function
PMQGMO	Pointer	Pointer to a data structure of type MQGMO
PMQHCONFIG	Pointer	Pointer to data of type MQHCONFIG
PMQHCONN	Pointer	Pointer to data of type MQHCONN
PMQHMSG	Pointer	Pointer to data of type MQHMSG
PMQHOBJ	Pointer	Pointer to data of type MQHOBJ
PMQIIH	Pointer	Pointer to a data structure of type MQIIH
PMQIMPO	Pointer	Pointer to a data structure of type MQIMPO
PMQINT8	Pointer	Pointer to data of type MQINT8
PMQINT16	Pointer	Pointer to data of type MQINT16
PMQINT32	Pointer	Pointer to data of type MQINT32
PMQINT64	Pointer	Pointer to data of type MQINT64
PMQLONG	Pointer	Pointer to data of type MQLONG
PMQMD	Pointer	Pointer to structure of type MQMD
PMQMDE	Pointer	Pointer to a data structure of type MQMDE
PMQMD1	Pointer	Pointer to a data structure of type MQMD1
PMQMD2	Pointer	Pointer to a data structure of type MQMD2
PMQMHBO	Pointer	Pointer to a data structure of type MQMHBO
PMQOD	Pointer	Pointer to a data structure of type MQOD
PMQOR	Pointer	Pointer to a data structure of type MQOR
PMQPD	Pointer	Pointer to a data structure of type MQPD
PMQPID	Pointer	Pointer to a process identifier
PMQMD	Pointer	Pointer to a data structure of type MQMD
PMQPMO	Pointer	Pointer to a data structure of type MQPMO

Elementary data type name	Data type	Description
PMQPTR	Pointer	Pointer to data of type MQPTR
PMQRFH	Pointer	Pointer to a data structure of type MQRFH
PMQRFH2	Pointer	Pointer to a data structure of type MQRFH2
PMQRMH	Pointer	Pointer to a data structure of type MQRMH
PMQRR	Pointer	Pointer to a data structure of type MQRR
PMQSCO	Pointer	Pointer to a data structure of type MQSCO
PMQSD	Pointer	Pointer to a data structure of type MQSD
PMQSMPO	Pointer	Pointer to a data structure of type MQSMPO
PMQSRO	Pointer	Pointer to a data structure of type MQSRO
PMSSTS	Pointer	Pointer to a data structure of type MQSTS
PMQTID	Pointer	Pointer to a thread ID
PMQTM	Pointer	Pointer to a data structure of type MQTM
PMQTC2	Pointer	Pointer to a data structure of type MQTC2
PMQUINT8	Pointer	Pointer to a data type of MQUINT8
PMQUINT16	Pointer	Pointer to a data type of MQUINT16
PMQUINT32	Pointer	Pointer to a data type of MQUINT32
PMQUINT64	Pointer	Pointer to a data type of MQUINT64
PMQULONG	Pointer	Pointer to a data type of MQULONG
PMQVOID	Pointer	
PMQWIH	Pointer	Pointer to a data structure of type MQWIH
PMQXQH	Pointer	Pointer to a data structure of type MQXQH

C declarations

Data type	Representation
MQBOOL	<code>typedef MQLONG MQBOOL;</code>
MQBYTE	<code>typedef unsigned char MQBYTE;</code>
MQBYTE8	<code>typedef MQBYTE MQBYTE8[8];</code>
MQBYTE16	<code>typedef MQBYTE MQBYTE16[16];</code>
MQBYTE24	<code>typedef MQBYTE MQBYTE24[24];</code>
MQBYTE32	<code>typedef MQBYTE MQBYTE32[32];</code>
MQBYTE40	<code>typedef MQBYTE MQBYTE40[40];</code>
MQCHAR	<code>typedef char MQCHAR;</code>
MQCHAR4	<code>typedef MQCHAR MQCHAR4[4];</code>
MQCHAR8	<code>typedef MQCHAR MQCHAR8[8];</code>
MQCHAR12	<code>typedef MQCHAR MQCHAR12[12];</code>
MQCHAR20	<code>typedef MQCHAR MQCHAR20[20];</code>
MQCHAR28	<code>typedef MQCHAR MQCHAR28[28];</code>
MQCHAR32	<code>typedef MQCHAR MQCHAR32[32];</code>
MQCHAR48	<code>typedef MQCHAR MQCHAR48[48];</code>
MQCHAR64	<code>typedef MQCHAR MQCHAR64[64];</code>
MQCHAR128	<code>typedef MQCHAR MQCHAR128[128];</code>
MQCHAR256	<code>typedef MQCHAR MQCHAR256[256];</code>
MQFLOAT32	<code>typedef float MQFLOAT32;</code>
MQFLOAT64	<code>typedef double MQFLOAT64;</code>
MQHCONFIG	<code>typedef void MQPOINTER MQHCONFIG;</code>

Data type	Representation
MQHCONN	<pre>typedef MQLONG MQHCONN;</pre>
MQHOBJ	<pre>typedef MQLONG MQHOBJ;</pre>
MQINT8	<pre>typedef signed char MQINT8;</pre>
MQINT16	<pre>typedef short MQINT16;</pre>
MQINT64	<p>On 64-bit UNIX systems:</p> <pre>typedef long;</pre> <p>On 32-bit AIX, Solaris, and HP-UX:</p> <pre>typedef int64_t;</pre> <p>On IBM i, Linux, and z/OS:</p> <pre>typedef long long;</pre> <p>On Windows:</p> <pre>typedef _int64;</pre>
MQLONG	<p>On IBM i:</p> <pre>typedef long MQLONG;</pre> <p>other platforms:</p> <pre>if defined(MQ_64_BIT) typedef int MQLONG; else typedef long MQLONG;</pre>
MQPID	<pre>typedef MQLONG MQPID;</pre>
MQPTR	<pre>typedef void MQPOINTER MQPTR;</pre>
MQTID	<pre>typedef MQLONG MQTID;</pre>
MQUINT8	<pre>typedef unsigned char MQUINT8;</pre>
MQUINT16	<pre>typedef unsigned short MQUINT16;</pre>

Data type	Representation
MQUINT64	<p>On 64-bit UNIX systems:</p> <pre>typedef unsigned long;</pre> <p>On 32-bit AIX, Solaris, and HP-UX:</p> <pre>typedef uint64_t;</pre> <p>On IBM i, Linux, and z/OS:</p> <pre>typedef unsigned long long;</pre> <p>On Windows:</p> <pre>typedef unsigned _int64;</pre>
MQULONG	<p>On IBM i:</p> <pre>typedef unsigned long MQULONG;</pre> <p>other platforms:</p> <pre>if defined(MQ_64_BIT) typedef unsigned int MQULONG; else typedef unsigned long MQULONG;</pre>
PMQBO	<pre>typedef MQBO MQPOINTER PMQBO;</pre>
PMQBOOL	<pre>typedef MQBOOL MQPOINTER PMQBOOL;</pre>
PMQBYTE	<pre>typedef MQBYTE MQPOINTER PMQBYTE;</pre>
PMQBYTE8	<pre>typedef MQBYTE8[8] MQPOINTER PMQBYTE8[8];</pre>
PMQBYTE16	<pre>typedef MQBYTE16[16] MQPOINTER PMQBYTE16[16];</pre>
PMQBYTE24	<pre>typedef MQBYTE24[24] MQPOINTER PMQBYTE24[24];</pre>
PMQBYTE32	<pre>typedef MQBYTE32[32] MQPOINTER PMQBYTE32[32];</pre>
PMQBYTE40	<pre>typedef MQBYTE40[40] MQPOINTER PMQBYTE40[40];</pre>
PMQBYTE128	<pre>typedef MQBYTE128[128] MQPOINTER PMQBYTE128[128];</pre>
PMQCHAR	<pre>typedef MQCHAR MQPOINTER PMQCHAR;</pre>
PMQCHAR4	<pre>typedef MQCHAR4[4] MQPOINTER PMQCHAR4[4];</pre>
PMQCHAR8	<pre>typedef MQCHAR8[8] MQPOINTER PMQCHAR8[8];</pre>

Data type	Representation
PMQCHAR12	<code>typedef MQCHAR12[12] MQPOINTER PMQCHAR12[12];</code>
PMQCHAR20	<code>typedef MQCHAR20[20] MQPOINTER PMQCHAR20[20];</code>
PMQCHAR28	<code>typedef MQCHAR28[28] MQPOINTER PMQCHAR28[28];</code>
PMQCHAR32	<code>typedef MQCHAR32[32] MQPOINTER PMQCHAR32[32];</code>
PMQCHAR48	<code>typedef MQCHAR48[48] MQPOINTER PMQCHAR48[48];</code>
PMQCHAR64	<code>typedef MQCHAR64[64] MQPOINTER PMQCHAR64[64];</code>
PMQCHAR128	<code>typedef MQCHAR128[128] MQPOINTER PMQCHAR128[128];</code>
PMQCHAR256	<code>typedef MQCHAR256[256] MQPOINTER PMQCHAR256[256];</code>
PMQCHAR264	<code>typedef MQCHAR264[264] MQPOINTER PMQCHAR264[264];</code>
PMQCIH	<code>typedef MQCIH MQPOINTER PMQCIH;</code>
PMQCNO	<code>typedef MQCNO MQPOINTER PMQCNO;</code>
PMQDLH	<code>typedef MQDLH MQPOINTER PMQDLH;</code>
PMQFUNC	<code>typedef void MQPOINTER PMQFUNC;</code>
PMQFLOAT32	<code>typedef MQFLOAT32 MQPOINTER PMQFLOAT32;</code>
PMQFLOAT64	<code>typedef MQFLOAT64 MQPOINTER PMQFLOAT64;</code>
PMQGMO	<code>typedef MQGMO MQPOINTER PMQGMO;</code>
PMQHCONFIG	<code>typedef MQHCONFIG MQPOINTER PMQHCONFIG;</code>
PMQHCONN	<code>typedef MQHCONN MQPOINTER PMQHCONN;</code>
PMQHOBJ	<code>typedef MQHOBJS MQPOINTER PMQHOBJS;</code>
PMQIIH	<code>typedef MQIIH MQPOINTER PMQIIH;</code>
PMQINT8	<code>typedef MQINT8 MQPOINTER PMQINT8;</code>

Data type	Representation
PMQINT16	<code>typedef MQINT16 MQPOINTER PMQINT16;</code>
PMQLONG	<code>typedef MQLONG MQPOINTER PMQLONG;</code>
PMQMD	<code>typedef MQMD MQPOINTER PMQMD;</code>
PMQMD1	<code>typedef MQMD1[1] MQPOINTER PMQMD1[1];</code>
PMQMDE	<code>typedef MQMDE MQPOINTER PMQMDE;</code>
PMQOD	<code>typedef MQOD MQPOINTER PMQOD;</code>
PMQPMO	<code>typedef MQPMO MQPOINTER PMQPMO;</code>
PMQPTR	<code>typedef MQPTR MQPOINTER PMQPTR;</code>
PMQRFH	<code>typedef MQRFH MQPOINTER PMQRFH;</code>
PMQRFH2	<code>typedef MQRFH2[2] MQPOINTER PMQRFH2[2];</code>
PMQRMH	<code>typedef MQRMH MQPOINTER PMQRMH;</code>
PMQTM	<code>typedef MQTM MQPOINTER PMQTM;</code>
PMQTM2	<code>typedef MQTMC2[2] MQPOINTER PMQTM2[2];</code>
PMQUINT8	<code>typedef MQUINT8 MQPOINTER PMQUINT8;</code>
PMQUINT16	<code>typedef MQUINT16 MQPOINTER PMQUINT16;</code>
PMQULONG	<code>typedef MQULONG MQPOINTER PMQULONG;</code>
PMQVOID	<code>typedef void MQPOINTER PMQVOID;</code>
PMQWIH	<code>typedef MQWIH MQPOINTER PMQWIH;</code>
PMQXQH	<code>typedef MQXQH MQPOINTER PMQXQH;</code>
PPMQBO	<code>typedef PMQBO MQPOINTER PPMQBO;</code>
PPMQBYTE	<code>typedef PMQBYTE MQPOINTER PPMQBYTE;</code>

Data type	Representation
PPMQCHAR	typedef PMQCHAR MQPOINTER PPMQCHAR;
PPMQCNO	typedef PMQCNO MQPOINTER PPMQCNO;
PPMQGMO	typedef PMQGMO MQPOINTER PPMQGMO;
PPMQHCONN	typedef PMQHCONN MQPOINTER PPMQHCONN;
PPMQHOBJ	typedef PMQHOBJ MQPOINTER PPMQHOBJ;
PPMQLONG	typedef PMLONG MQPOINTER PPMQLONG;
PPMQMD	typedef PMQMD MQPOINTER PPMQMD;
PPMQOD	typedef PMQOD MQPOINTER PPMQOD;
PPMQPMO	typedef PMQPMO MQPOINTER PPMQPMO;
PPMQULONG	typedef PMQULONG MQPOINTER PPMQULONG;
PPMQVOID	typedef PMQVOID MQPOINTER PPMQVOID;
Where defined (MQ_64_BIT) means a 64 bit platform.	

See “Data types” on page 256 for a description of the MQPOINTER macro variable.

COBOL declarations

Data type	Representation
MQBOOL	PIC S9(9) BINARY
MQBYTE	PIC X
MQBYTE8	PIC X(8)
MQBYTE16	PIC X(16)
MQBYTE24	PIC X(24)
MQBYTE32	PIC X(32)
MQBYTE40	PIC X(40)

Data type	Representation
MQCHAR	PIC X
MQCHAR4	PIC X(4)
MQCHAR8	PIC X(8)
MQCHAR12	PIC X(12)
MQCHAR20	PIC X(20)
MQCHAR28	PIC X(28)
MQCHAR32	PIC X(32)
MQCHAR48	PIC X(48)
MQCHAR64	PIC X(64)
MQCHAR128	PIC X(128)
MQCHAR256	PIC X(256)
MQFLOAT32	USAGE COMP-1
MQFLOAT64	USAGE COMP-2
MQHCONN	PIC S9(9) BINARY
MQHOBJ	PIC S9(9) BINARY
MQINT8	PIC S9(2) BINARY
MQINT16	PIC S9(4) BINARY
MQINT64	PIC S9(18) BINARY
MQLONG	PIC S9(9) BINARY
MQPTR	POINTER
MQUINT8	PIC 9(2) BINARY

Data type	Representation
MQUINT16	PIC 9(4) BINARY
MQUINT64	PIC 9(18) BINARY
MQULONG	PIC 9(9) BINARY

PL/I declarations

PL/I is supported on z/OS.

Data type	Representation
MQBOOL	fixed bin(31)
MQBYTE	char(1)
MQBYTE8	char(8)
MQBYTE16	char(16)
MQBYTE24	char(24)
MQBYTE32	char(32)
MQBYTE40	char(40)
MQCHAR	char(1)
MQCHAR4	char(4)
MQCHAR8	char(8)
MQCHAR12	char(12)
MQCHAR20	char(20)
MQCHAR28	char(28)
MQCHAR32	char(32)
MQCHAR48	char(48)
MQCHAR64	char(64)

Data type	Representation
MQCHAR128	char(128)
MQCHAR256	char(256)
MQFLOAT32	binary float(21) ieee
MQFLOAT64	binary float(52) ieee
MQHCONN	fixed bin(31)
MQHOBJ	fixed bin(31)
MQINT8	fixed bin(7)
MQINT16	fixed bin(15)
MQINT64	fixed bin(63)
MQLONG	fixed bin(31)
MQPTR	pointer
MQUINT8	fixed bin(8)
MQUINT16	fixed bin(16)
MQUINT64	fixed bin(64)
MQULONG	fixed bin(32)

System/390 assembler declarations
System/390 assembler is supported on z/OS only.

Data type	Representation
MQBOOL	DS F
MQBYTE	DS XL1
MQBYTE8	DS XL8
MQBYTE16	DS XL16

Data type	Representation
MQBYTE24	DS XL24
MQBYTE32	DS XL32
MQBYTE40	DS XL40
MQCHAR	DS CL1
MQCHAR4	DS CL4
MQCHAR8	DS CL8
MQCHAR12	DS CL12
MQCHAR20	DS CL20
MQCHAR28	DS CL28
MQCHAR32	DS CL32
MQCHAR48	DS CL48
MQCHAR64	DS CL64
MQCHAR128	DS CL128
MQCHAR256	DS CL256
MQFLOAT32	DS EB
MQFLOAT64	DS DB
MQHCONN	DS F
MQHOBJ	DS F
MQINT8	DS XL1
MQINT16	DS H
MQINT64	DS D

Data type	Representation
MQLONG	DS F
MQPTR	DS F
MQUINT8	DS XL1
MQUINT16	DS H
MQUINT64	DS D
MQULONG	DS F

Structure data types - introduction

This section introduces the structure data types used in the MQI. The structure data types themselves are described in subsequent sections.

Summary

The following tables summarize the structure data types used in the MQI.

<i>Table 456. Structure data types used on MQI calls (or exit functions):</i>		
Structure	Description	Calls where used
MQACH	API exit chain header	
MQAIR	Authentication information record	MQCONN
MQAXC	API exit context	
MQAXP	API exit parameter	
MQBMHO	Buffer to message handle options	MQBUFMH
MQBO	Begin options	MQBEGIN
MQCBD	Callback descriptor	MQCB
MQCBO	Create-bag options	mqCreateBag
MQCHARV	Variable length string	MQINQMP
MQCNO	Connect options	MQCONN
MQCSP	Security parameters	MQCONN
MQCTLO	Callback options	MQCTL
MQDMPO	Delete message property options	MQDLTMP
MQGMO	Get-message options	MQGET
MQIMPO	Inquire message property options	MQINQMP

Table 456. Structure data types used on MQI calls (or exit functions): (continued)

Structure	Description	Calls where used
MQMD	Message descriptor	MQBUFMH , MQMHBUF , MQCB , MQGET , MQPUT , MQPUT1
MQMHBO	Message handle to buffer options	MQMHBUF
MQOD	Object descriptor	MQOPEN , MQPUT1
MQOR	Object record	MQOPEN , MQPUT1
MQPD	Property descriptor	MQSETMP
MQPMO	Put-message options	MQPUT , MQPUT1
MQPMR	Put-message record	MQPUT , MQPUT1
MQRR	Response record	MQOPEN , MQPUT , MQPUT1
MQSCO	SSL configuration options	MQCONN
MQSD	Subscription descriptor	MQSUB
MQSMPO	Set message property option	MQSETMP
MQSRO	Subscription request options	MQSUBRQ
MQSTS	Status reporting structure	MQSTAT

Table 457. Structure data types used in message data:

Structure	Description
MQCIH	CICS information header
MQCFH	PCF header
MQEPH	Embedded PCF header
MQDH	Distribution header
MQDLH	Dead letter (undelivered message) header
MQIIH	IMS information header
MQMDE	Message descriptor extension
MQRFH	Rules and formatting header
MQRFH2	Rules and formatting header 2
MQRMH	Reference message header
MQTM	Trigger message
MQTMC2	Trigger message (character format 2)
MQWIH	Work information header
MQXQH	Transmission queue header

Note: The MQDXP structure (data conversion exit parameter) is described in [“Data-conversion exit”](#) on page 870, together with the associated data conversion calls.

Rules for structure data types

Programming languages vary in their level of support for structures, and certain rules and conventions are adopted to map the MQI structures consistently in each programming language:

1. Structures must be aligned on their natural boundaries.
 - Most MQI structures require 4-byte alignment.
 - On IBM i, structures containing pointers require 16-byte alignment; these are: MQCNO, MQOD, MQPMO.
2. Each field in a structure must be aligned on its natural boundary.
 - Fields with data types that equate to MQLONG must be aligned on 4-byte boundaries.
 - Fields with data types that equate to MQPTR must be aligned on 16-byte boundaries on IBM i, and 4-byte boundaries in other environments.
 - Other fields are aligned on 1-byte boundaries.
3. The length of a structure must be a multiple of its boundary alignment.
 - Most MQI structures have lengths that are multiples of 4 bytes.
 - On IBM i, structures containing pointers have lengths that are multiples of 16 bytes.
4. Where necessary, padding bytes or fields must be added to ensure compliance with the preceding rules.

Conventions used in the descriptions

The description of each structure data type includes:

- An overview of the purpose and use of the structure
- Descriptions of the fields in the structure, in a form that is independent of the programming language
- Examples of how the structure is declared in each of the supported programming languages

The description of each structure data type contains the following sections:

Structure name

The name of the structure, followed by a summary of the fields in the structure.

Overview

A brief description of the purpose and use of the structure.

Fields

Descriptions of the fields. For each field, the name of the field is followed by its elementary data type in parentheses (). In text, field names are shown using an italic typeface; for example *Version*.

There is also a description of the purpose of the field, together with a list of any values that the field can take. Names of constants are shown in uppercase; for example MQGMO_STRUC_ID. A set of constants having the same prefix is shown using the * character, for example: MQIA_*.

In the descriptions of the fields, the following terms are used:

input

You supply information in the field when you make a call.

output

The queue manager returns information in the field when the call completes or fails.

input/output

You supply information in the field when you make a call, and the queue manager changes the information when the call completes or fails.

Initial values

A table showing the initial values for each field in the data definition files supplied with the MQI.

C declaration

Typical declaration of the structure in C.

COBOL declaration

Typical declaration of the structure in COBOL.

PL/I declaration

Typical declaration of the structure in PL/I.

System/390 assembler declaration

Typical declaration of the structure in System/390 assembler language.

Visual Basic declaration

Typical declaration of the structure in Visual Basic.



C programming

This section contains information to help you use the MQI from the C programming language.

Header files

Header files are provided to help you write C application programs that use the MQI.

These header files are summarized in [Table 458 on page 255](#).

Table 458. C header files	
File	Contents
CMQC	Function prototypes, data types, and named constants for the main MQI
CMQXC	Function prototypes, data types, and named constants for the data conversion exit
CMQEC	Function prototypes, data types, and named constants for the main MQI, data conversion exit and Interface Entry Points structure (CMQEC includes CMQXC and CMQC.)
CMQSTRC	Functions that convert MQI constant definitions to the text equivalent.  Attention:  Not applicable to z/OS.

To improve the portability of applications, code the name of the header file in lowercase on the `#include` preprocessor directive:

```
#include "cmqec.h"
```

Functions

You do not need to specify all parameters that are passed by address every time you invoke a function.

- Pass parameters that are *input-only* and of type MQHCONN, MQHOBJ, or MQLONG by value.
- Pass all other parameters by address.

Where a particular parameter is not required, use a null pointer as the parameter on the function invocation, in place of the address of the parameter data. Parameters for which this is possible are identified in the call descriptions.

No parameter is returned as the value of the function; in C terminology, this means that all functions return `void`.

The attributes of the function are defined by the MQENTRY macro variable; the value of this macro variable depends on the environment.

Parameters with undefined data type

The *Buffer* parameter on the MQGET, MQPUT, and MQPUT1 functions has an undefined data type. This parameter is used to send and receive the application's message data.

Parameters of this sort are shown in the C examples as arrays of MQBYTE. You can declare the parameters in this way, but it is usually more convenient to declare them as the particular structure

that describes the layout of the data in the message. Declare the actual function parameter as a pointer-to-void, and specify the address of any sort of data as the parameter on the function invocation.

Data types

Define all data types using the C typedef statement. For each data type, also define the corresponding pointer data type. The name of the pointer data type is the name of the elementary or structure data type prefixed with the letter P to denote a pointer. Define the attributes of the pointer using the MQPOINTER macro variable; the value of this macro variable depends on the environment. The following illustrates how to declare pointer data types:

```
#define MQPOINTER *          /* depends on environment */
...
typedef MQLONG MQPOINTER PMQLONG; /* pointer to MQLONG */
typedef MQMD MQPOINTER PMQMD; /* pointer to MQMD */
```

Manipulating binary strings

Declare strings of binary data as one of the MQBYTEn data types.

Whenever you copy, compare, or set fields of this type, use the C functions memcpy , memcmp , or memset ; for example:

```
#include <string.h>
#include "cmqc.h"

MQMD MyMsgDesc;

memcpy(MyMsgDesc.MsgId,          /* set "MsgId" field to nulls */
       MQMI_NONE,              /* ...using named constant */
       sizeof(MyMsgDesc.MsgId));

memset(MyMsgDesc.CorrelId,       /* set "CorrelId" field to nulls */
       0x00,                   /* ...using a different method */
       sizeof(MQBYTE24));
```

Do not use the string functions strcpy , strcmp , strncpy , or strncmp , because these do not work correctly for data declared with the MQBYTEn data types.

Manipulating character strings

When the queue manager returns character data to the application, the queue manager always pads the character data with blanks to the defined length of the field; the queue manager *does not* return null-terminated strings.

Therefore, when copying, comparing, or concatenating such strings, use the string functions strncpy , strncmp , or strncat .

Do not use the string functions that require the string to be terminated by a null (strcpy , strcmp , strcat). Also, do not use the function strlen to determine the length of the string; use instead the sizeof function to determine the length of the field.

Initial values for structures

The header files define various macro variables that you can use to provide initial values for the MQ structures when you declare instances of those structures.

These macro variables have names of the form MQxxx_DEFAULT, where MQxxx represents the name of the structure. They are used in the following way:

```
MQMD MyMsgDesc = {MQMD_DEFAULT};
MQPMO MyPutOpts = {MQPMO_DEFAULT};
```

For some character fields (for example, the *StrucId* fields that occur in most structures, or the *Format* field that occurs in MQMD), the MQI defines particular values that are valid. For each of the valid values, two macro variables are provided:

- One macro variable defines the value as a string with a length, excluding the implied null matches, exactly the defined length of the field. For example, for the *Format* field in MQMD the following macro variable is provided (␣ represents a blank character):

```
#define MQFMT_STRING "MQSTR␣␣␣"
```

Use this form with the `memcpy` and `memcmp` functions.

- The other macro variable defines the value as an array of characters; the name of this macro variable is the name of the string form suffixed with `_ARRAY`. For example:

```
#define MQFMT_STRING_ARRAY 'M','Q','S','T','R','␣','␣','␣'
```

Use this form to initialize the field when you declare an instance of the structure with values different from those provided by the `MQMD_DEFAULT` macro variable. (This is not always necessary; in some environments you can use the string form of the value in both situations. However, you can use the array form for declarations, because this is required for compatibility with the C++ programming language.)

Initial values for dynamic structures

When a variable number of instances of a structure is required, the instances are typically created in main storage obtained dynamically using the `calloc` or `malloc` functions. To initialize the fields in such structures, consider the following technique:

1. Declare an instance of the structure using the appropriate `MQxxx_DEFAULT` macro variable to initialize the structure. This instance becomes the model for other instances:

```
MQMD Model = {MQMD_DEFAULT}; /* declare model instance */
```

The `static` or `auto` keywords can be coded on the declaration in order to give the model instance static or dynamic lifetime, as required.

2. Use the `calloc` or `malloc` functions to obtain storage for a dynamic instance of the structure:

```
PMQMD Instance;  
Instance = malloc(sizeof(MQMD)); /* get storage for dynamic instance */
```

3. Use the `memcpy` function to copy the model instance to the dynamic instance:

```
memcpy(Instance,&Model,sizeof(MQMD)); /* initialize dynamic instance */
```

Use from C++

For the C++ programming language, the header files contain the following additional statements that are included only when you use a C++ compiler:

```
#ifndef __cplusplus  
extern "C" {  
#endif  
  
/* rest of header file */  
  
#ifdef __cplusplus  
}  
#endif
```

Notational conventions

This information shows how to invoke the functions and declare parameters.

In some cases, the parameters are arrays with a size that is not fixed. For these, a lowercase `n` is used to represent a numeric constant. When you code the declaration for that parameter, replace the `n` with the numeric value required.

COBOL programming

This section contains information to help you use the MQI from the COBOL programming language.

COPY files

Various COPY files are provided to help you write COBOL application programs that use the MQI. There are two files containing named constants, and two files for each of the structures.

Each structure is provided in two forms: a form with initial values, and a form without:

- Use the structures with initial values in the WORKING-STORAGE SECTION of a COBOL program; they are contained in COPY files with names suffixed with the letter V (for Values).
- Use the structures without initial values in the LINKAGE SECTION of a COBOL program; they are contained in COPY files with names suffixed with the letter L (for Linkage).

The COPY files are summarized in [Table 459 on page 258](#). Not all the files listed are available in all environments.

<i>Table 459. COBOL COPY files</i>		
File (with initial values)	File (without initial values)	Contents
CMQAIRV	CMQAIRL	Authentication information record
CMQBOV	CMQBOL	Begin options structure
CMQCIHV	CMQCIHL	CICS information header structure
CMQCN OV	CMQCNOL	Connect options structure
CMQD HV	CMQDHL	Distribution header structure
CMQDLHV	CMQDLHL	Dead letter header structure
CMQDXPV	CMQDXPL	Data conversion exit parameter structure
CMQGM OV	CMQGMOL	Get message options structure
CMQIIHV	CMQIIHL	IMS information header structure
CMQMDV	CMQMDL	Message descriptor structure
CMQMDEV	CMQMDEL	Message descriptor extension structure
CMQMD1V	CMQMD1L	Message descriptor structure version 1
CMQODV	CMQODL	Object descriptor structure
CMQORV	CMQORL	Object record structure
CMQPM OV	CMQPMOL	Put message options structure
CMQRFHV	CMQRFHL	Rules and formatting header structure
CMQRFH2V	CMQRFH2L	Rules and formatting header structure version 2
CMQRMHV	CMQRMHL	Reference message header structure
CMQRRV	CMQRRL	Response record structure
CMQSCOV	CMQSCOL	SSL configuration options
CMQTMV	CMQTML	Trigger message structure
CMQTM CV	CMQTMCL	Trigger message structure (character format)
CMQTM C2V	CMQTM C2L	Trigger message structure (character format) version 2
CMQWIHV	CMQWIHL	Work information header structure
CMQXQHV	CMQXQHL	Transmission queue header structure

Table 459. COBOL COPY files (continued)		
File (with initial values)	File (without initial values)	Contents
CMQV	-	Named constants for main MQI
CMQXV	-	Named constants for data conversion exit
CMQMD2V	CMQMD2L	Message descriptor structure version 2

Structures

In the COPY file, each structure declaration begins with a level-10 item; this enables you to declare several instances of the structure by coding the level-01 declaration and then using the COPY statement to copy in the remainder of the structure declaration. To refer to the appropriate instance, use the IN keyword:

```
* Declare two instances of MQMD
01 MY-MQMD.
   COPY CMQMDV.
01 MY-OTHER-MQMD.
   COPY CMQMDV.
*
* Set MSGTYPE field in MY-OTHER-MQMD
  MOVE MQMT-REQUEST TO MQMD-MSGTYPE IN MY-OTHER-MQMD.
```

Align the structures on appropriate boundaries. If you use the COPY statement to include a structure following an item that is not the level-01 item, ensure that the structure begins at the appropriate offset from the start of the level-01 item. Most MQI structures require 4-byte alignment; the exceptions to this are MQCNO, MQOD, and MQPMO, which require 16-byte alignment on IBM i.

In this section, the names of fields in structures are shown without a prefix. In COBOL, the field names are prefixed with the name of the structure followed by a hyphen. However, if the structure name ends with a numeric digit, indicating that the structure is a second or later version of the original structure, the numeric digit is omitted from the prefix. Field names in COBOL are shown in uppercase (although lowercase or mixed case can be used if required). For example, the field *MsgType* described in [“MQMD - Message descriptor”](#) on page 396 becomes MQMD-MSGTYPE in COBOL.

The V-suffix structures are declared with initial values for all the fields; you need to set only those fields where you want a value that is different from the supplied initial value.

Pointers

Some structures need to address optional data that might be discontinuous with the structure, such as the MQOR and MQRR records addressed by the MQOD structure.

To address this optional data, the structures contain fields that are declared with the pointer data type. However, COBOL does not support the pointer data type in all environments. Because of this, the optional data can also be addressed using fields that contain the offset of the data from the start of the structure.

If you want to port an application between environments, ascertain whether the pointer data type is available in all the intended environments. If it is not, the application must address the optional data using the offset fields instead of the pointer fields.

In those environments where pointers are not supported, declare the pointer fields as byte strings of the appropriate length, with the initial value being the all-null byte string. Do not alter this initial value if you are using the offset fields.

Named constants

In this section, the names of constants are shown containing the underscore character (_) as part of the name. In COBOL, use the hyphen character (-) in place of the underscore.

Constants that have character-string values use the single quotation mark as the string delimiter ('). In some environments, you might have to specify an appropriate compiler option to cause the compiler to accept the single quotation mark as the string delimiter in place of the double quotation mark.

The named constants are declared in the COPY files as level-10 items. To use the constants, declare the level-01 item explicitly, and then use the COPY statement to copy in the declarations of the constants:

```
* Declare a structure to hold the constants
01 MY-MQ-CONSTANTS.
   COPY CMQV.
```

The preceding method causes the constants to occupy storage in the program even if they are not referenced. If you include the constants in many separate programs within the same run unit, multiple copies of the constants exist, consuming main storage unnecessarily. Avoid this effect by using one of the following techniques:

- Add the GLOBAL clause to the level-01 declaration:

```
* Declare a global structure to hold the constants
01 MY-MQ-CONSTANTS GLOBAL.
   COPY CMQV.
```

This causes allocates storage for only one set of constants within the run unit. The constants, however, can be referenced by any program within the run unit, not just the program that contains the level-01 declaration.

Note: The GLOBAL clause is not supported in all environments.

- Manually copy into each program only those constants that are referenced by that program. Do not use the COPY statement to copy all the constants into the program.

Notational conventions

The latter topics in this section show how to invoke the calls and declare parameters. In some cases, the parameters are tables or character strings the size of which is not fixed. For these, a lowercase n is used to represent a numeric constant. When you code the declaration for that parameter, replace the n with the numeric value required.

System/390 assembler programming

This section contains information to help to you use the MQI from the System/390 Assembler programming language.

Macros

Various macros are provided to help you to write assembler application programs that use the MQI.

There are two macros for named constants, and one macro for each of the structures. These files are summarized in [Table 460 on page 260](#).

<i>Table 460. Assembler macros</i>	
File	Contents
CMQA	Named constants (equates) for main MQI
CMQCIHA	CICS information header structure
CMQCNOA	Connect options structure
CMQDLHA	Dead letter header structure
CMQDXPA	Data conversion exit parameter structure
CMQGMOA	Get message options structure
CMQIIHA	IMS information header structure
CMQMDA	Message descriptor structure
CMQMDEA	Message descriptor extension structure
CMQODA	Object descriptor structure

Table 460. Assembler macros (continued)	
File	Contents
CMQPMOA	Put message options structure
CMQRFHA	Rules and formatting header structure
CMQRFH2A	Rules and formatting header structure version 2
CMQRMHA	Reference message header structure
CMQTMA	Trigger message structure
CMQTMCA	Trigger message structure (character format) version 2
CMQVERA	Structure version control
CMQWIHA	Work information header structure
CMQXA	Named constants for data conversion exit
CMQXPA	API crossing exit parameter structure
CMQXQHA	Transmission queue header structure

Structures

The structures are generated by macros that have various parameters to control the action of the macro. These parameters are described in the following sections.

From time to time new versions of the MQ structures are introduced. The additional fields in a new version can cause a structure that previously was smaller than 256 bytes to become larger than 256 bytes. Because of this, write assembler instructions that are intended to copy an MQ structure, or to set an MQ structure to nulls, to work correctly with structures that might be larger than 256 bytes. Alternatively, use the DCLVER macro parameter or CMQVERA macro with the VERSION parameter to declare a specific version of the structure.

Specifying the name of the structure

To declare more than one instance of a structure, the macro prefixes the name of each field in the structure with a user-specifiable string and an underscore.

The string used is the label specified on the invocation of the macro. If no label is specified, the name of the structure is used to construct the prefix:

```
* Declare two object descriptors
      CMQODA      ,      Prefix used="MQOD_" (the default)
MY_MQOD CMQODA  ,      Prefix used="MY_MQOD_"
```

The structure declarations shown in this section use the default prefix.

Specifying the form of the structure

Structure declarations can be generated by the macro in one of two forms, controlled by the DSECT parameter:

DSECT=YES

An assembler DSECT instruction is used to start a new data section; the structure definition immediately follows the DSECT statement. The label on the macro invocation is used as the name of the data section; if no label is specified, the name of the structure is used.

DSECT=NO

Assembler DC instructions are used to define the structure at the current position in the routine. The fields are initialized with values, which can be specified by coding the relevant parameters on the macro invocation. Fields for which no values are specified on the macro invocation are initialized with default values.

The value specified must be uppercase. If the DSECT parameter is not specified, DSECT=NO is assumed.

Controlling the version of the structure

By default, the macros always declare the most recent version of each structure.

Although you can use the VERSION macro parameter to specify a value for the *Version* field in the structure, that parameter defines the initial value for the *Version* field, and does not control the version of the structure actually declared. To control the version of the structure that is declared, use the DCLVER parameter:

DCLVER=CURRENT

The version declared is the current (most recent) version.

DCLVER=SPECIFIED

The version declared is the version specified by the VERSION parameter. If you omit the VERSION parameter, the default is version 1.

If you specify the VERSION parameter, the value must be a self-defining numeric constant, or the named constant for the version required (for example, MQCNO_VERSION_3). If you specify some other value, the structure is declared as if DCLVER=CURRENT had been specified, even if the value of VERSION resolves to a valid value.

The value specified must be uppercase. If you omit the DCLVER parameter, the value used is taken from the MQDCLVER global macro variable. You can set this variable using the CMQVERA macro.

Declaring one structure embedded within another

To declare one structure as a component of another structure, use the NESTED parameter:

NESTED=YES

The structure declaration is nested within another.

NESTED=NO

The structure declaration is not nested within another.

The value specified must be uppercase. If you omit the NESTED parameter, NESTED=NO is assumed.

Specifying initial values for fields

Specify the value to be used to initialize a field in a structure by coding the name of that field (without the prefix) as a parameter on the macro invocation, accompanied by the value required.

For example, to declare a message-descriptor structure with the *MsgType* field initialized with MQMT_REQUEST, and the *ReplyToQ* field initialized with the string "MY_REPLY_TO_QUEUE", use the following:

```
MY_MQMD  CMQMDA  MSGTYPE=MQMT_REQUEST,          X
          REPLYTOQ=MY_REPLY_TO_QUEUE
```

If you specify a named constant (equate) as a value on the macro invocation, use the CMQA macro to define the named constant. Do not enclose character string values in single quotation marks.

Controlling the listing

Control the appearance of the structure declaration in the assembler listing using the LIST parameter:

LIST=YES

The structure declaration appears in the assembler listing.

LIST=NO

The structure declaration does not appear in the assembler listing.

The value specified must be uppercase. If you omit the LIST parameter, LIST=NO is assumed.

CMQVERA macro

This macro allows you to set the default value to be used for the DCLVER parameter on the structure macros. The value specified by CMQVERA is used by the structure macro only if you omit the DCLVER parameter from the invocation of the structure macro. The default value is set by coding the CMQVERA macro with the DCLVER parameter:

DCLVER=CURRENT

The default version is set to the current (most recent) version.

DCLVER=SPECIFIED

The default version is set to the version specified by the VERSION parameter.

You must specify the DCLVER parameter, and the value must be uppercase. The value set by CMQVERA remains the default value until the next invocation of CMQVERA, or the end of the assembly. If you omit CMQVERA, the default is DCLVER=CURRENT .

Notational conventions

Later sections show how to invoke the calls and declare parameters. In some cases, the parameters are arrays or character strings with a size that is not fixed for which, a lowercase n is used to represent a numeric constant. When you code the declaration for that parameter, replace the n with the numeric value required.

MQAIR - Authentication information record

The MQAIR structure represents the authentication information record.

The following table summarizes the fields in the structure.

Table 461. Fields in MQAIR		
Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
AuthInfoType	Type of authentication information	AuthInfoType
AuthInfoConnName	Connection name of LDAP CRL server	AuthInfoConnName
LDAPUserNamePtr	Address of LDAP user name	LDAPUserNamePtr
LDAPUserNameOffset	Offset of LDAP user name from start of MQSCO	LDAPUserNameOffset
LDAPUserNameLength	Length of LDAP user name	LDAPUserNameLength
LDAPPassword	Password to access LDAP server	LDAPPassword
Note: The remaining fields are ignored if <i>Version</i> is less than MQAIR_VERSION_2.		
OCSPResponderURL	URL at which the OCSP responder can be contacted	OCSPResponderURL

Overview for MQAIR

The MQAIR structure allows an application running as an IBM MQ MQI client to specify information about an authenticator that is to be used for the client connection. The structure is an input parameter on the MQCONN call.

Availability: AIX, HP-UX, Solaris, Linux and Windows clients.

Character set and encoding: Data in MQAIR must be in the character set and encoding of the local queue manager; these are given by the **CodedCharSetId** queue manager attribute and MQENC_NATIVE.

Fields for MQAIR

The MQAIR structure contains the following fields; the fields are described in **alphabetical order**:

AuthInfoConnName (MQCHAR264)

This is either the host name or the network address of a host on which the LDAP server is running. This can be followed by an optional port number, enclosed in parentheses. The default port number is 389.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. If the value is not valid, the call fails with reason code MQRC_AUTH_INFO_CONN_NAME_ERROR.

This is an input field. The length of this field is given by MQ_AUTH_INFO_CONN_NAME_LENGTH. The initial value of this field is the null string in C, and blank characters in other programming languages.

AuthInfoType (MQLONG)

This is the type of authentication information contained in the record.

The value can be one of the two following parameters:

MQAIT_CRL_LDAP

Certificate revocation checking using LDAP server.

MQAIT_OCSP

Certificate revocation checking using OCSP.

If the value is not valid, the call fails with reason code MQRC_AUTH_INFO_TYPE_ERROR.

This is an input field. The initial value of this field is MQAIT_CRL_LDAP.

LDAPPassword (MQCHAR32)

This is the password needed to access the LDAP CRL server. If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field.

If the LDAP server does not require a password, or you omit the LDAP user name, *LDAPPassword* must be null or blank. If you omit the LDAP user name and *LDAPPassword* is not null or blank, the call fails with reason code MQRC_LDAP_PASSWORD_ERROR.

This is an input field. The length of this field is given by MQ_LDAP_PASSWORD_LENGTH. The initial value of this field is the null string in C, and blank characters in other programming languages.

LDAPUserNameLength (MQLONG)

This is the length in bytes of the LDAP user name addressed by the *LDAPUserNamePtr* or *LDAPUserNameOffset* field. The value must be in the range zero through MQ_DISTINGUISHED_NAME_LENGTH. If the value is not valid, the call fails with reason code MQRC_LDAP_USER_NAME_LENGTH_ERR.

If the LDAP server involved does not require a user name, set this field to zero.

This is an input field. The initial value of this field is 0.

LDAPUserNameOffset (MQLONG)

This is the offset in bytes of the LDAP user name from the start of the MQAIR structure.

The offset can be positive or negative. The field is ignored if *LDAPUserNameLength* is zero.

You can use either *LDAPUserNamePtr* or *LDAPUserNameOffset* to specify the LDAP user name, but not both; see the description of the *LDAPUserNamePtr* field for details.

This is an input field. The initial value of this field is 0.

LDAPUserNamePtr (PMQCHAR)

This is the LDAP user name.

It consists of the Distinguished Name of the user who is attempting to access the LDAP CRL server. If the value is shorter than the length specified by *LDAPUserNameLength*, terminate the value with a null character, or pad it with blanks to the length *LDAPUserNameLength*. The field is ignored if *LDAPUserNameLength* is zero.

You can supply the LDAP user name in one of two ways:

- By using the pointer field *LDAPUserNamePtr*

In this case, the application can declare a string that is separate from the MQAIR structure, and set *LDAPUserNamePtr* to the address of the string.

Consider using *LDAPUserNamePtr* for programming languages that support the pointer data type in a fashion that is portable to different environments (for example, the C programming language).

- By using the offset field *LDAPUserNameOffset*

In this case, the application must declare a compound structure containing the MQSCO structure followed by the array of MQAIR records followed by the LDAP user name strings, and set *LDAPUserNameOffset* to the offset of the appropriate name string from the start of the MQAIR structure. Ensure that this value is correct, and has a value that can be accommodated within an MQLONG (the most restrictive programming language is COBOL, for which the valid range is -999 999 999 through +999 999 999).

Consider using *LDAPUserNameOffset* for programming languages that do not support the pointer data type, or that implement the pointer data type in a fashion that might not be portable to different environments (for example, the COBOL programming language).

Whichever technique is chosen, use only one of *LDAPUserNamePtr* and *LDAPUserNameOffset*; the call fails with reason code MQRC_LDAP_USER_NAME_ERROR if both are nonzero.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

OCSPResponderURL (MQCHAR256)

For an MQAIR structure that represents connection details for an OCSP responder, this field contains the URL at which the responder can be contacted.

The value of this field is an HTTP URL. This field takes priority over a URL in an AuthorityInfoAccess (AIA) certificate extension.

The value is ignored unless both the following statements are true:

- The MQAIR structure is Version 2 or later (the Version field is set to MQAIR_VERSION_2 or greater).
- The AuthInfoType field is set to MQAIT_OCSP.

If the field does not contain an HTTP URL in the correct format (and is not being ignored), the MQCONN call fails with reason code MQRC_OCSP_URL_ERROR.

This field is case-sensitive. It must start with the string `http://` in lowercase. The rest of the URL might be case-sensitive, depending on the OCSP server implementation.

This field is not subject to data conversion.

StrucId (MQCHAR4)

The value must be:

MQAIR_STRUC_ID

Identifier for the authentication information record.

For the C programming language, the constant `MQAIR_STRUC_ID_ARRAY` is also defined; this has the same value as `MQAIR_STRUC_ID`, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is `MQAIR_STRUC_ID`.

Version (MQLONG)

The version number of the MQAIR structure.

The value must be one of the following:

MQAIR_VERSION_1

Version-1 authentication information record.

MQAIR_VERSION_2

Version-2 authentication information record.

The following constant specifies the version number of the current version:

MQAIR_CURRENT_VERSION

Current version of authentication information record.

This is always an input field. The initial value of this field is MQAIR_VERSION_1.

Initial values and language declarations for MQAIR

Table 462. Initial values of fields in MQAIR		
Field name	Name of constant	Value of constant
StrucId	MQAIR_STRUC_ID	'AIR¬'
Version	MQAIR_VERSION_1	1
AuthInfoType	MQAIT_CRL_LDAP	1
AuthInfoConnName	None	Null string or blanks
LDAPUserNamePtr	None	Null pointer or null bytes
LDAPUserNameOffset	None	0
LDAPUserNameLength	None	0
LDAPPassword	None	Null string or blanks
OCSPResponderURL	None	Null string or blanks
Notes: 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQAIR_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQAIR MyAIR = {MQAIR_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQAIR MQAIR;
struct tagMQAIR {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     AuthInfoType;     /* Type of authentication
                                information */
    MQCHAR264  AuthInfoConnName; /* Connection name of CRL LDAP
                                server */
    PMQCHAR    LDAPUserNamePtr;  /* Address of LDAP user name */
    MQLONG     LDAPUserNameOffset; /* Offset of LDAP user name from start
                                of MQAIR structure */
    MQLONG     LDAPUserNameLength; /* Length of LDAP user name */
    MQCHAR32   LDAPPassword;     /* Password to access LDAP server */
    MQCHAR256  OCSPResponderURL; /* URL of OCSP responder */
};
```

COBOL declaration

```
**    MQAIR structure
10  MQAIR.
**      Structure identifier
15  MQAIR-STRUCID          PIC X(4).
```

```

**      Structure version number
15 MQAIR-VERSION          PIC S9(9) BINARY.
**      Type of authentication information
15 MQAIR-AUTHINFOTYPE     PIC S9(9) BINARY.
**      Connection name of CRL LDAP server
15 MQAIR-AUTHINFOCONNNAME PIC X(264).
**      Address of LDAP user name
15 MQAIR-LDAPUSERNAMEPTR  POINTER.
**      Offset of LDAP user name from start of MQAIR structure
15 MQAIR-LDAPUSERNAMEOFFSET PIC S9(9) BINARY.
**      Length of LDAP user name
15 MQAIR-LDAPUSERNAMELENGTH PIC S9(9) BINARY.
**      Password to access LDAP server
15 MQAIR-LDAPPASSWORD     PIC X(32).
**      URL of OSCP responder
15 MQAIR-OCSPRESPONDERURL PIC X(256).

```

Visual Basic declaration

```

Type MQAIR
  StrucId          As String*4   'Structure identifier'
  Version          As Long       'Structure version number'
  AuthInfoType     As Long       'Type of authentication information'
  AuthInfoConnName As String*264 'Connection name of CRL LDAP server'
  LDAPUserNamePtr  As MQPTR      'Address of LDAP user name'
  LDAPUserNameOffset As Long     'Offset of LDAP user name from start'
                                'of MQAIR structure'
  LDAPUserNameLength As Long     'Length of LDAP user name'
  LDAPPASSWORD     As String*32  'Password to access LDAP server'
End Type

```

MQBMHO - Buffer to message handle options

The following table summarizes the fields in the structure. MQBMHO structure - buffer to message handle options

Table 463. Fields in MQBMHO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options controlling the action of MQBMHO	Options

Overview for MQBMHO

Availability: All. Buffer to message handle options structure - overview

Purpose: The MQBMHO structure allows applications to specify options that control how message handles are produced from buffers. The structure is an input parameter on the MQBUFMH call.

Character set and encoding: Data in MQBMHO must be in the character set of the application and encoding of the application (MQENC_NATIVE).

Fields for MQBMHO

Buffer to message handle options structure - fields

The MQBMHO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

Buffer to message handle structure - Options field

The value can be:

MQBMHO_DELETE_PROPERTIES

Properties that are added to the message handle are deleted from the buffer. If the call fails no properties are deleted.

Default options: If you do not need the option described, use the following option:

MQBMHO_NONE

No options specified.

This is always an input field. The initial value of this field is MQBMHO_DELETE_PROPERTIES.

StrucId (MQCHAR4)

Buffer to message handle structure - StrucId field

This is the structure identifier. The value must be:

MQBMHO_STRUC_ID

Identifier for buffer to message handle structure.

For the C programming language, the constant MQBMHO_STRUC_ID_ARRAY is also defined; this has the same value as MQBMHO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQBMHO_STRUC_ID.

Version (MQLONG)

Buffer to message handle structure - Version field

This is the structure version number. The value must be:

MQBMHO_VERSION_1

Version number for buffer to message handle structure.

The following constant specifies the version number of the current version:

MQBMHO_CURRENT_VERSION

Current version of buffer to message handle structure.

This is always an input field. The initial value of this field is MQBMHO_VERSION_1.

Initial values and language declarations for MQBMHO

Buffer to message handle structure - Initial values

Table 464. Initial values of fields in MQBMHO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQBMHO_STRUC_ID	' BMHO '
<i>Version</i>	MQBMHO_VERSION_1	1
<i>Options</i>	MQBMHO_NONE	0
Notes:		
1. In the C programming language, the macro variable MQBMHO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:		
<pre>MQBMHO MyBMHO = {MQBMHO_DEFAULT};</pre>		

C declaration

Buffer to message handle structure - C language declaration

```
typedef struct tagMQBMHO MQBMHO;
struct tagMQBMHO {
    MQCHAR4  StrucId;          /* Structure identifier */
    MQLONG   Version;         /* Structure version number */
    MQLONG   Options;         /* Options that control the action of
                             MQBUFMH */
};
```

COBOL declaration

Buffer to message handle structure - COBOL language declaration

```
** MQBMHO structure
10 MQBMHO.
**   Structure identifier
15 MQBMHO-STRUCID          PIC X(4).
**   Structure version number
15 MQBMHO-VERSION          PIC S9(9) BINARY.
**   Options that control the action of MQBUFMH
15 MQBMHO-OPTIONS          PIC S9(9) BINARY.
```

PL/I declaration

Buffer to message handle structure - PL/I language declaration

```
Dcl
1 MQBMHO based,
3 StrucId      char(4),      /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 Options      fixed bin(31), /* Options that control the action
                             of MQBUFMH */
```

High Level Assembler declaration

Buffer to message handle structure - Assembler language declaration

```
MQBMHO          DSECT
MQBMHO_STRUCID   DS   CL4   Structure identifier
MQBMHO_VERSION   DS   F     Structure version number
MQBMHO_OPTIONS   DS   F     Options that control the
*                  action of MQBUFMH
MQBMHO_LENGTH    EQU   *-MQBMHO
MQBMHO_AREA      DS   CL(MQBMHO_LENGTH)
```

MQBO - Begin options

The following table summarizes the fields in the structure.

Table 465. Fields in MQBO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options that control the action of MQBEGIN	Options

Overview for MQBO

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows ; not available for IBM MQ MQI clients.

Purpose: The MQBO structure allows the application to specify options relating to the creation of a unit of work. The structure is an input/output parameter on the MQBEGIN call.

Character set and encoding: Data in MQBO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQBO

The MQBO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

This field is always an input field. Its initial value is MQBO_NONE.

The value must be:

MQBO_NONE

No options specified.

StrucId (MQCHAR4)

This field is always an input field. Its initial value is MQBO_STRUC_ID.

The value must be:

MQBO_STRUC_ID

Identifier for begin-options structure.

For the C programming language, the constant MQBO_STRUC_ID_ARRAY is also defined; this has the same value as MQBO_STRUC_ID, but is an array of characters instead of a string.

Version (MQLONG)

This field is always an input field. Its initial value is MQBO_VERSION_1.

The value must be:

MQBO_VERSION_1

Version number for begin-options structure.

The following constant specifies the version number of the current version:

MQBO_CURRENT_VERSION

Current version of begin-options structure.

Initial values and language declarations for MQBO

Table 466. Initial values of fields in MQBO for MQBO		
Field name	Name of constant	Value of constant
StrucId	MQBO_STRUC_ID	'B0¬¬'
Version	MQBO_VERSION_1	1
Options	MQBO_NONE	0
Notes: 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQBO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQBO MyBO = {MQBO_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQBO MQBO;  
struct tagMQBO {  
    MQCHAR4 StrucId; /* Structure identifier */  
    MQLONG Version; /* Structure version number */  
    MQLONG Options; /* Options that control the action of MQBEGIN */  
};
```

COBOL declaration

```
** MQBO structure  
10 MQBO.  
** Structure identifier  
15 MQBO-STRUCID PIC X(4).  
** Structure version number  
15 MQBO-VERSION PIC S9(9) BINARY.  
** Options that control the action of MQBEGIN  
15 MQBO-OPTIONS PIC S9(9) BINARY.
```

PL/I declaration

```
dcl
  1 MQBO based,
    3 StrucId char(4),      /* Structure identifier */
    3 Version fixed bin(31), /* Structure version number */
    3 Options fixed bin(31); /* Options that control the action of
                               MQBEGIN */
```

Visual Basic declaration

```
Type MQBO
  StrucId As String*4 'Structure identifier'
  Version As Long     'Structure version number'
  Options As Long     'Options that control the action of MQBEGIN'
End Type
```

MQCBC - Callback context

The following table summarizes the fields in the structure. Structure describing the callback routine.

Table 467. Fields in MQCBC		
Field	Description	Topic
<i>StrucID</i>	Structure identifier	StrucID
<i>Version</i>	Structure version number	Version
<i>CallType</i>	Why function has been called	CallType
<i>Hobj</i>	Object handle	Hobj
<i>CallbackArea</i>	Field for callback function to use	CallbackArea
<i>ConnectionArea</i>	Field for callback function to use	ConnectionArea
<i>CompCode</i>	Completion code	CompCode
<i>Reason</i>	Reason code	Reason
<i>State</i>	Indication of the state of the current consumer	State
<i>DataLength</i>	Message length	DataLength
<i>BufferLength</i>	Length of message buffer in bytes	BufferLength
<i>Flags</i>	General flags	Flags
Note: The remaining field is ignored if Version is less than MQCBC_VERSION_2		
<i>ReconnectDelay</i>	Number of milliseconds before reconnection attempt	ReconnectDelay

Overview for MQCBC

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQCBC structure is used to specify context information that is passed to a callback function.

The structure is an input/output parameter on the call to a message consumer routine.

Version: The current version of MQCBC is MQCBC_VERSION_2.

Character set and encoding: Data in MQCBC must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure will be in the character set and encoding of the client.

Fields for MQCBC

Alphabetic list of fields for the MQCBC structure.

The MQCBC structure contains the following fields; the fields are described in alphabetical order:

BufferLength (MQLONG)

This field is the length in bytes of the message buffer that has been passed to this function.

The buffer can be larger than both the MaxMsgLength value defined for the consumer and the ReturnedLength value in the MQGMO.

The actual message length is supplied in [DataLength](#) field.

The application can use the entire buffer for its own purposes for the duration of the callback function.

This is an input field to the message consumer function; it is not relevant to an exception handler function.

CallbackArea (MQPTR)

This field is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the [CallbackArea](#) field in the MQCBD structure, which is a parameter on the MQCB call used to define the callback function.

Changes to the *CallbackArea* are preserved across the invocations of the callback function for an *HObj*. This field is not shared with callback functions for other handles.

This is an input/output field to the callback function. The initial value of this field is a null pointer or null bytes.

CallType (MQLONG)

Field containing information about why this function has been called; the following are defined.

Message delivery call types: These call types contain information about a message. The *DataLength* and *BufferLength* parameters are valid for these call types.

MQCBCT_MSG_REMOVED

The message consumer function has been invoked with a message that has been destructively removed from the object handle.

If the value of *CompCode* is MQCC_WARNING, the value of the *Reason* field is MQRC_TRUNCATED_MSG_ACCEPTED or one of the codes indicating a data conversion problem.

MQCBCT_MSG_NOT_REMOVED

The message consumer function has been invoked with a message that has not yet been destructively removed from the object handle. The message can be destructively removed from the object handle using the *MsgToken*.

The message might not have been removed because:

- The MQGMO options requested a browse operation, MQGMO_BROWSE_*
- The message is larger than the available buffer and the MQGMO options do not specify MQGMO_ACCEPT_TRUNCATED_MSG

If the value of *CompCode* is MQCC_WARNING, the value of the *Reason* field is MQRC_TRUNCATED_MSG_FAILED or one of the codes indicating a data conversion problem.

Callback control call types: These call types contain information about the control of the callback and do not contain details about a message. These call types are requested using Options in the MQCBD structure.

The *DataLength* and *BufferLength* parameters are not valid for these call types.

MQCBCT_REGISTER_CALL

The purpose of this call type is to allow the callback function to perform some initial setup.

The callback function is invoked immediately after the callback is registered, that is, upon return from an MQCB call using a value for the *Operation* field of MQOP_REGISTER.

This call type is used both for message consumers and event handlers.

If requested, this is the first invocation of the callback function.

The value of the *Reason* field is MQRC_NONE.

MQCBCT_START_CALL

The purpose of this call type is to allow the callback function to perform some setup when it is started, for example, reinstating resources that were cleaned up when it was previously stopped.

The callback function is invoked when the connection is started using either MQOP_START or MQOP_START_WAIT.

If a callback function is registered within another callback function, this call type is invoked when the callback returns.

This call type is used for message consumers only.

The value of the *Reason* field is MQRC_NONE.

MQCBCT_STOP_CALL

The purpose of this call type is to allow the callback function to perform some cleanup when it is stopped for a while, for example, cleaning up additional resources that have been acquired during the consuming of messages.

The callback function is invoked when an MQCTL call is issued using a value for the *Operation* field of MQOP_STOP.

This call type is used for message consumers only.

The value of the *Reason* field is set to indicate the reason for stopping.

MQCBCT_DEREGISTER_CALL

The purpose of this call type is to allow the callback function to perform final cleanup at the end of the consume process. The callback function is invoked when the:

- Callback function is deregistered using an MQCB call with MQOP_DEREGISTER.
- Queue is closed, causing an implicit deregister. In this instance the callback function is passed MQHO_UNUSABLE_HOBJ as the object handle.
- MQDISC call completes - causing an implicit close and, therefore, a deregister. In this case the connection is not disconnected immediately, and any ongoing transaction is not yet committed.

If any of these actions are taken inside the callback function itself, the action is invoked once the callback returns.

This call type is used both for message consumers and event handlers.

If requested, this is the last invocation of the callback function.

The value of the *Reason* field is set to indicate the reason for stopping.

MQCBCT_EVENT_CALL

Event handler function

The event handler function has been invoked without a message when the queue manager or connection stops or quiesces.

This call can be used to take appropriate action for all callback functions.

Message consumer function

The message consumer function has been invoked without a message when an error (*CompCode* = MQCC_FAILED) has been detected that is specific to the object handle; for example *Reason* code = MQRC_GET_INHIBITED.

The value of the *Reason* field is set to indicate the reason for the call.

MQCBCT_MC_EVENT_CALL

The event handler function has been invoked for multicast events; The event handler is sent IBM MQ Multicast events instead of 'normal' IBM MQ events.

For more information about MQCBCT_MC_EVENT_CALL, see [Multicast exception reporting](#).

CompCode (MQLONG)

This field is the completion code. It indicates whether there were any problems consuming the message.

The value is one of the following:

MQCC_OK

Successful completion

MQCC_WARNING

Warning (partial completion)

MQCC_FAILED

Call failed

This is an input field. The initial value of this field is MQCC_OK.

ConnectionArea (MQPTR)

This field is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the [ConnectionArea](#) field in the MQCTLO structure, which is a parameter on the MQCTL call used to control the callback function.

Any changes made to this field by the callback functions are preserved across the invocations of the callback function. This area can be used to pass information that is to be shared by all callback functions. Unlike *CallbackArea*, this area is common across all callbacks for a connection handle.

This is an input and output field. The initial value of this field is a null pointer or null bytes.

DataLength (MQLONG)

This is the length in bytes of the application data in the message. If the value is zero, it means that the message contains no application data.

The DataLength field contains the length of the message but not necessarily the length of the message data passed to the consumer. It could be that the message was truncated. Use the [ReturnedLength](#) field in the MQGMO to determine how much data has actually been passed to the consumer.

If the reason code indicates the message has been truncated, you can use the DataLength field to determine how large the actual message is. This allows you to determine the size of the buffer required to accommodate the message data, and then issue an MQCB call to update the [MaxMsgLength](#) with an appropriate value.

If the MQGMO_CONVERT option is specified, the converted message could be larger than the value returned for DataLength. In such cases, the application probably needs to issue an MQCB call to update the [MaxMsgLength](#) to be greater than the value returned by the queue manager for DataLength.

To avoid message truncation problems, specify MaxMsgLength as MQCBD_FULL_MSG_LENGTH. This causes the queue manager to allocate a buffer for the full message length after data conversion. Be aware, however, that even if this option is specified, it is still possible that sufficient storage is not

available to correctly process the request. Applications should always check the returned reason code. For example, if it is not possible to allocate sufficient storage to convert the message, the messages is returned to the application unconverted.

This is an input field to the message consumer function; it is not relevant to an event handler function.

Flags (MQLONG)

Flags containing information about this consumer.

The following option is defined:

MQCBCF_READA_BUFFER_EMPTY

This flag can be returned if a previous MQCLOSE call using the MQCO_QUIESCE option failed with a reason code of MQRC_READ_AHEAD_MSGS.

This code indicated that the last read ahead message is being returned and that the buffer is now empty. If the application issues another MQCLOSE call using the MQCO_QUIESCE) option, it succeeds.

Note, that an application is not guaranteed to be given a message with this flag set, as there might still be messages in the read-ahead buffer that do not match the current selection criteria. In this instance, the consumer function is invoked with the reason code MQRC_HOBJ_QUIESCED.

If the read ahead buffer is completely empty, the consumer is invoked with the MQCBCF_READA_BUFFER_EMPTY flag and the reason code MQRC_HOBJ_QUIESCED_NO_MSGS.

This is an input field to the message consumer function; it is not relevant to an event handler function.

Hobj (MQHOBJ)

This is the object handle for calls to the message consumer.

For an event handler, this value is MQHO_NONE

The application can use this handle and the message token in the Get Message Options block to get the message if a message has not been removed from the queue.

This is always an input field. The initial value of this field is MQHO_UNUSABLE_HOBJ

Reason (MQLONG)

This is the reason code qualifying the *CompCode*.

This is an input field. The initial value of this field is MQRC_NONE.

State (MQLONG)

An indication as to the state of the current consumer. This field is of most value to an application when a nonzero reason code is passed to the consumer function.

You can use this field to simplify application programming because you do not need to code behavior for each reason code.

This is an input field. The initial value of this field is MQCS_NONE

State	Queue manager action	Value of constant
<i>MQCS_NONE</i> This reason code represents a normal call with no additional reason information	None; this is the normal operation.	0

State	Queue manager action	Value of constant
<i>MQCS_SUSPENDED_TEMPORARY</i> These reason codes represent temporary conditions.	The callback routine is called to report the condition and then suspended. After a period of time the system might attempt the operation again, which can lead to the same condition being raised again.	1
<i>MQCS_SUSPENDED_USER_ACTION</i> These reason codes represent conditions where the callback needs to take action to resolve the condition.	The consumer is suspended and the callback routine is called to report the condition. The callback routine should resolve the condition if possible and either RESUME or close down the connection.	2
<i>MQCS_SUSPENDED</i> These reason codes represent failures that prevent further message callbacks.	The queue manager automatically suspends the callback function. If the callback function is resumed it is likely to receive the same reason code again.	3
<i>MQCS_STOPPED</i> These reason codes represent the end of message consumption.	Delivered to the exception handler and to callbacks that specified MQCBDO_STOP_CALL. No further messages can be consumed.	4

StrucId (MQCHAR4)

The value in this field is the structure identifier.

The value must be:

MQCBC_STRUC_ID

Identifier for callback context structure.

For the C programming language, the constant MQCBC_STRUC_ID_ARRAY is also defined; this has the same value as MQCBC_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQCBC_STRUC_ID.

Version (MQLONG)

The value in this field is the structure version number.

The value must be:

MQCBC_VERSION_1

Version-1 callback context structure.

The following constant specifies the version number of the current version:

MQCBC_CURRENT_VERSION

Current version of the callback context structure.

This is always an input field. The initial value of this field is MQCBC_VERSION_1.

The callback function is always passed the latest version of the structure.

ReconnectDelay (MQLONG)

ReconnectDelay indicates how long the queue manager will wait before trying to reconnect. The field can be modified by an event handler to change the delay or stop reconnection altogether.

Use the ReconnectDelay field only if the value of the Reason field in the Callback Context is MQRC_RECONNECTING.

On entry to the event handler the value of ReconnectDelay is the number of milliseconds the queue manager is going to wait before making a reconnection attempt. Table 468 on page 277 lists the values that you can set to modify the behavior of the queue manager on return from the event handler.

Table 468. ReconnectDelay values

Name	Value	Description
MQRD_NO_RECONNECT	-1	Make no more reconnection attempts. An error is returned to the application.
MQRD_NO_DELAY	0	Try to reconnect immediately.
Milliseconds	>0	Wait for this many milliseconds before retrying the connection.

Initial values and language declarations for MQCBC

Callback context structure - initial values

There are no initial values for the **MQCBC** structure. The structure is passed as a parameter to a callback routine. The queue manager initializes the structure; applications never initialize it.

C declaration

Callback context structure - C language declaration

```
typedef struct tagMQCBC MQCBC;
struct tagMQCBC {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     CallType;         /* Why Function was called */
    MQHOBj     Hobj;            /* Object Handle */
    MQPTR      CallbackArea;     /* Callback data passed to the function */
    MQPTR      ConnectionArea;   /* MQCTL data area passed to the function */
    MQLONG     CompCode;         /* Completion Code */
    MQLONG     Reason;           /* Reason Code */
    MQLONG     State;            /* Consumer State */
    MQLONG     DataLength;       /* Message Data Length */
    MQLONG     BufferLength;     /* Buffer Length */
    MQLONG     Flags;            /* Flags containing information about
                                this consumer */

    /* Ver:1 */
    MQLONG     ReconnectDelay;   /* Number of milliseconds before */
    /* Ver:2 */ }               /* reconnect attempt */
```

COBOL declaration

```
** MQCBC structure
10  MQCBC.
** Structure Identifier
15  MQCBC-STRUCID                PIC X(4).
** Structure Version
15  MQCBC-VERSION                PIC S9(9) BINARY.
** Call Type
15  MQCBC-CALLTYPE               PIC S9(9) BINARY.
** Object Handle
15  MQCBC-HOBJ                   PIC S9(9) BINARY.
** Callback User Area
15  MQCBC-CALLBACKAREA           POINTER
** Connection Area
15  MQCBC-CONNECTIONAREA         POINTER
** Completion Code
15  MQCBC-COMPCODE               PIC S9(9) BINARY.
** Reason Code
15  MQCBC-REASON                 PIC S9(9) BINARY.
** Consumer State
15  MQCBC-STATE                  PIC S9(9) BINARY.
** Data Length
15  MQCBC-DATALENGTH             PIC S9(9) BINARY.
** Buffer Length
15  MQCBC-BUFFERLENGTH           PIC S9(9) BINARY.
** Flags
15  MQCBC-FLAGS                  PIC S9(9) BINARY.
** Ver:1 **
** Number of milliseconds before reconnect attempt
15  MQCBC-RECONNECTDELAY PIC S9(9) BINARY.
** Ver:2 **
```

PL/I declaration

```
dcl
  1 MQCBC based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version */
    3 CallType         fixed bin(31),    /* Callback type */
    3 Hobj             fixed bin(31),    /* Object Handle */
    3 CallbackArea     pointer,          /* User area passed to the function */
    3 ConnectionArea   pointer,          /* Connection User Area */
    3 CompCode         fixed bin(31);    /* Completion Code */
    3 Reason           fixed bin(31);    /* Reason Code */
    3 State            fixed bin(31);    /* Consumer State */
    3 DataLength       fixed bin(31);    /* Message Data Length */
    3 BufferLength     fixed bin(31);    /* Message Buffer length */
    3 Flags            fixed bin(31);    /* Consumer Flags */
/* Ver:1 */
    3 ReconnectDelay   fixed bin(31);    /* Number of milliseconds before */
/* Ver:2 */              /* reconnect attempt */
```

High Level Assembler declaration

```
MQCBC          DSECT
MQCBC          DS 0F      Force fullword alignment
MQCBC_STRUCID   DS CL4    Structure identifier
MQCBC_VERSION   DS F      Structure version number
MQCBC_CALLTYPE  DS F      Why Function was called
MQCBC_HOBJ      DS F      Object Handle
MQCBC_CALLBACKAREA DS A    Callback data passed to the function
MQCBC_CONNECTIONAREA DS A  MQCTL Data area passed to the function
MQCBC_COMPCODE  DS F      Completion Code
MQCBC_REASON    DS F      Reason Code
MQCBC_STATE     DS F      Consumer State
MQCBC_DATALENGTH DS F      Message Data Length
MQCBC_BUFFERLENGTH DS F  Buffer Length
MQCBC_FLAGS     DS F      Flags containing information about this consumer
MQCBC_RECONNECTDELAY DS F  Number of milliseconds before reconnect
MQCBC_LENGTH    EQU *-MQCBC
                ORG      MQCBC
MQCBC_AREA      DS CL(MQCBC_LENGTH)
```

MQCBD - Callback descriptor

The following table summarizes the fields in the structure. Structure specifying the callback function.

Table 469. Fields in MQCBD		
Field	Description	Topic
<i>StrucID</i>	Structure identifier	StrucID
<i>Version</i>	Structure version number	Version
<i>CallbackType</i>	Type of callback function	CallbackType
<i>Options</i>	Options controlling message consumption	Options
<i>Callback Area</i>	Field for callback function to use	CallbackArea
<i>CallbackFunction</i>	Whether the function is invoked as an API call	CallbackFunction
<i>CallbackName</i>	Whether the function is invoked as a dynamically-linked program	CallbackName
<i>MaxMsgLength</i>	Length of longest message that can be read	MaxMsgLength

Overview for MQCBD

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS, and IBM MQ MQI clients connected to these systems.

Purpose: The MQCBD structure is used to specify a callback function and the options controlling its use by the queue manager.

The structure is an input parameter on the MQCB call.

Version: The current version of MQCBD is MQCBD_VERSION_1.

Character set and encoding: Data in MQCBD must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQCBD

Alphabetic list of fields for the MQCBD structure.

The MQCBD structure contains the following fields; the fields are described in alphabetical order:

CallbackArea (MQPTR)

Callback descriptor structure - CallbackArea field

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the [CallbackArea](#) field in the MQCBC structure, which is a parameter on the callback function declaration.

The value is used only on an *Operation* having a value MQOP_REGISTER, with no currently defined callback, it does not replace a previous definition.

This is an input and output field to the callback function. The initial value of this field is a null pointer or null bytes.

CallbackFunction (MQPTR)

Callback descriptor structure - CallbackFunction field

The callback function is invoked as a function call.

Use this field to specify a pointer to the callback function.

You *must* specify either *CallbackFunction* or *CallbackName*. If you specify both, the reason code MQRC_CALLBACK_ROUTINE_ERROR is returned.

If neither *CallbackName* nor *CallbackFunction* is set, the call fails with the reason code MQRC_CALLBACK_ROUTINE_ERROR.

This option is not supported in the following environment: Programming languages and compilers that do not support function-pointer references. In such situations, the call fails with the reason code MQRC_CALLBACK_ROUTINE_ERROR.

On z/OS the function must expect to be called with OS linkage conventions. For example, in the C programming language, specify:

```
#pragma linkage(MQCB_FUNCTION,OS)
```

This is an input field. The initial value of this field is a null pointer or null bytes.

Note: When using CICS with IBM MQ V7.0.1, asynchronous consumption is supported if:

- Apar PK66866 is applied to CICS TS 3.2
- Apar PK89844 is applied to CICS TS 4.1

CallbackName (MQCHAR128)

Callback descriptor structure - CallbackName field

The callback function is invoked as a dynamically linked program.

You *must* specify either *CallbackFunction* or *CallbackName*. If you specify both, the reason code MQRC_CALLBACK_ROUTINE_ERROR is returned.

If neither *CallbackName* nor *CallbackFunction* is not set, the call fails with the reason code MQRC_CALLBACK_ROUTINE_ERROR.

The module is loaded when the first callback routine to use is registered, and unloaded when the last callback routine to use it deregisters.

Except where noted in the following text, the name is left-justified within the field, with no embedded blanks; the name itself is padded with blanks to the length of the field. In the descriptions that follow, square brackets ([]) denote optional information:

IBM i

The callback name can be one of the following formats:

- Library "/" Program
- Library "/" ServiceProgram ("FunctionName")

For example, MyLibrary/MyProgram(MyFunction).

The library name can be *LIBL. Both the library and program names are limited to a maximum of 10 characters.

UNIX systems

The callback name is the name of a dynamically-loadable module or library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path:

```
[path]library(function)
```

If the path is not specified the system search path is used.

The name is limited to a maximum of 128 characters.

Windows

The callback name is the name of a dynamic-link library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path and drive:

```
[d:][path]library(function)
```

If the drive and path are not specified the system search path is used.

The name is limited to a maximum of 128 characters.

z/OS

The callback name is the name of a load module that is valid for specification on the EP parameter of the LINK or LOAD macro.

The name is limited to a maximum of 8 characters.

z/OS CICS

The callback name is the name of a load module that is valid for specification on the PROGRAM parameter of the EXEC CICS LINK command macro.

The name is limited to a maximum of 8 characters.

The program can be defined as remote using the REMOTESYTEM option of the installed PROGRAM definition or by the dynamic routing program.

The remote CICS region must be connected to IBM MQ if the program is to use IBM MQ API calls.

Note, however, that the Hobj field in the MQCBC structure is not valid in a remote system.

If a failure occurs trying to load *CallbackName*, one of the following error codes is returned to the application:

- MQRC_MODULE_NOT_FOUND

- MQRC_MODULE_INVALID
- MQRC_MODULE_ENTRY_NOT_FOUND

A message is also written to the error log containing the name of the module for which the load was attempted, and the failing reason code from the operating system.

This is an input field. The initial value of this field is a null string or blanks.

CallbackType (MQLONG)

Callback descriptor structure - CallbackType field

This is the type of the callback function. The value must be one of:

MQCBT_MESSAGE_CONSUMER

Defines this callback as a message consumer function.

A message consumer callback function is called when a message, meeting the selection criteria specified, is available on an object handle and the connection is started.

MQCBT_EVENT_HANDLER

Defines this callback as the asynchronous event routine; it is not driven to consume messages for a handle.

Hobj is not required on the MQCB call defining the event handler and is ignored if specified.

The event handler is called for conditions that affect the whole message consumer environment. The consumer function is invoked without a message when an event, for example, a queue manager or connection stopping, or quiescing, occurs. It is not called for conditions that are specific to a single message consumer, for example, MQRC_GET_INHIBITED.

Events are delivered to the application, regardless of whether the connection is started or stopped, except in the following environments:

- CICS on z/OS environment
- nonthreaded applications

If the caller does not pass one of these values, the call fails with a *Reason* code of MQRC_CALLBACK_TYPE_ERROR

This is always an input field. The initial value of this field is MQCBT_MESSAGE_CONSUMER.

MaxMsgLength (MQLONG)

This is the length in bytes of the longest message that can be read from the handle and given to the callback routine. Callback descriptor structure - MaxMsgLength field

If a message has a longer length, the callback routine receives *MaxMsgLength* bytes of the message, and reason code:

- MQRC_TRUNCATED_MSG_FAILED or
- MQRC_TRUNCATED_MSG_ACCEPTED if you specified MQGMO_ACCEPT_TRUNCATED_MSG.

The actual message length is supplied in the DataLength field of the MQCBC structure.

The following special value is defined:

MQCBD_FULL_MSG_LENGTH

The buffer length is adjusted by the system to return messages without truncation.

If insufficient memory is available to allocate a buffer to receive the message, the system calls the callback function with an MQRC_STORAGE_NOT_AVAILABLE reason code.

If, for example, you request data conversion, and there is insufficient memory available to convert the message data, the unconverted message is passed to the callback function.

This is an input field. The initial value of the *MaxMsgLength* field is MQCBD_FULL_MSG_LENGTH.

Options (MQLONG)

Callback descriptor structure - Options field

You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

MQCBDO_FAIL_IF_QUIESCING

The MQCB call fails if the queue manager is in the quiescing state.

On z/OS, this option also forces the MQCB call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

Specify MQGMO_FAIL_IF_QUIESCING, in the MQGMO options passed on the MQCB call, to cause notification to message consumers when they are quiescing.

Control options: The following options control whether the callback function is called, without a message, when the state of the consumer changes:

MQCBDO_REGISTER_CALL

The callback function is invoked with call type MQCBCT_REGISTER_CALL.

MQCBDO_START_CALL

The callback function is invoked with call type MQCBCT_START_CALL.

MQCBDO_STOP_CALL

The callback function is invoked with call type MQCBCT_STOP_CALL.

MQCBDO_DEREGISTER_CALL

The callback function is invoked with call type MQCBCT_DEREGISTER_CALL.

MQCBDO_EVENT_CALL

The callback function is invoked with call type MQCBCT_EVENT_CALL.

MQCBDO_MC_EVENT_CALL

The callback function is invoked with call type MQCBCT__MC_EVENT_CALL.

See [CallType](#) for further details about these call types.

Default option: If you do not need any of the options described, use the following option:

MQCBDO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values.

MQCBDO_NONE is defined to aid program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *Options* field is MQCBDO_NONE.

StrucId (MQCHAR4)

Callback descriptor structure - StrucId field

This is the structure identifier; the value must be:

MQCBD_STRUC_ID

Identifier for callback descriptor structure.

For the C programming language, the constant MQCBD_STRUC_ID_ARRAY is also defined; this has the same value as MQCBD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQCBD_STRUC_ID.

Version (MQLONG)

Callback descriptor structure - Version field

This is the structure version number; the value must be:

MQCBD_VERSION_1

Version-1 callback descriptor structure.

The following constant specifies the version number of the current version:

MQCBD_CURRENT_VERSION

Current version of callback descriptor structure.

This is always an input field. The initial value of this field is MQCBD_VERSION_1.

Initial values and language declarations for MQCBD

Callback descriptor structure - Initial values

Table 470. Initial values of fields in MQCBD		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCBD_STRUC_ID	'CBD~'
<i>Version</i>	MQCBD_VERSION_1	1
<i>CallBackType</i>	MQCBT_MESSAGE_CONSUMER	1
<i>Options</i>	MQCBDO_NONE	0
<i>CallbackArea</i>	None	Null pointer or null blanks
<i>CallbackFunction</i>	None	Null pointer or null blanks
<i>CallbackName</i>	None	Null string or blanks
<i>MaxMsgLength</i>	MQCBD_FULL_MSG_LENGTH	-1
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null sting in the C programming language, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQCBD_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQCBD MyCBD = {MQCBD_DEFAULT};</pre> 		

C declaration

Callback descriptor structure - C language declaration

```
typedef struct tagMQCBD MQCBD;
struct tagMQCBD {
    MCHAR4    StrucId;           /* Structure identifier */
    MLONG     Version;           /* Structure version number */
    MLONG     CallBackType;      /* Callback function type */
    MLONG     Options;           /* Options controlling message
                                consumption */
    MQPTR     CallbackArea;      /* User data passed to the function */
    MQPTR     CallbackFunction;  /* Callback function pointer */
    MCHAR128  CallbackName;      /* Callback name */
    MLONG     MaxMsgLength;      /* Maximum message length */
};
```

COBOL declaration

```
** MQCBD structure
10  MQCBD.
** Structure Identifier
15  MQCBD-STRUCID                PIC X(4).
** Structure Version
15  MQCBD-VERSION                PIC S9(9) BINARY.
** Callback Type
15  MQCBD-CALLBACKTYPE           PIC S9(9) BINARY.
```

```

** Options
15 MQCBD-OPTIONS                PIC S9(9) BINARY.
** Callback User Area
15 MQCBD-CALLBACKAREA           POINTER
** Callback Function Pointer
15 MQCBD-CALLBACKFUNCTION       FUNCTION-POINTER
** Callback Program Name
15 MQCBD-CALLBACKNAME           PIC X(128)
** Maximum Message Length
15 MQCDB-MAXMSGLENGTH           PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQCBD based,
3 StrucId          char(4),          /* Structure identifier*/
3 Version          fixed bin(31),    /* Structure version*/
3 CallbackType     fixed bin(31),    /* Callback function type */
3 Options          fixed bin(31),    /* Options */
3 CallbackArea     pointer,          /* User area passed to the function */
3 CallbackFunction pointer,          /* Callback Function Pointer */
3 CallbackName     char(128),        /* Callback Program Name */
3 MaxMsgLength     fixed bin(31);    /* Maximum Message Length */

```

MQCHARV - Variable Length String

The following table summarizes the fields in the structure.

Field	Description	Topic
<i>VSPtr</i>	Pointer to the variable length string	VSPtr
<i>VSOffset</i>	Offset in bytes of the variable length string from the start of the structure that contains this MQCHARV structure	VSOffset
<i>VSLength</i>	The length in bytes of the variable length string addressed by the VSPtr or VSOffset field.	VSLength
<i>VSBufSize</i>	The size in bytes of the buffer addressed by the VSPtr or VSOffset field.	VSBufSize
<i>VSCCSID</i>	The character set identifier of the variable length string addressed by the VSPtr or VSOffset field.	VSCCSID

Overview for MQCHARV

Availability: AIX, HP-UX, Solaris, Linux, IBM i, Windows, plus IBM MQ MQI clients connected to these systems.

Purpose: Use the MQCHARV structure to describe a variable length string.

Character set and encoding: Data in the MQCHARV must be in the encoding of the local queue manager that is given by MQENC_NATIVE and the character set of the VSCCSID field within the structure. If the application is running as an MQ client, the structure must be in the encoding of the client. Some character sets have a representation that depends on the encoding. If VSCCSID is one of these character sets, the encoding used is the same encoding as that of the other fields in the MQCHARV. The character set identified by VSCCSID can be a double-byte character set (DBCS).

Usage: The MQCHARV structure addresses data that might be discontiguous with the structure containing it. To address this data, fields declared with the pointer data type can be used. Be aware that COBOL does not support the pointer data type in all environments. Because of this, the data can also be addressed using fields that contain the offset of the data from the start of the structure containing the MQCHARV.

COBOL programming

If you want to port an application between environments, you must ascertain whether the pointer data type is available in all the intended environments. If not, the application must address the data using the offset fields instead of the pointer fields.

In those environments where pointers are not supported, you can declare the pointer fields as byte strings of the appropriate length, with the initial value being the all-null byte string. Do not alter this initial value if you are using the offset fields. One way to do this without changing the supplied copy books is to use the following:

```
COPY CMQCHRVV REPLACING POINTER BY ==BINARY PIC S9(9)==.
```

where CMQCHRVV can be exchanged for the copy book to be used.

Fields for MQCHARV

The MQCHARV structure contains the following fields; the fields are described in **alphabetical order**:

VSBufSize (MQLONG)

This is the size in bytes of the buffer addressed by the VSPtr or VSOOffset field.

When the MQCHARV structure is used as an output field on a function call, this field must be initialised with the length of the buffer provided. If the value of VSLength is greater than VSBufSize then only VSBufSize bytes of data are returned to the caller in the buffer.

This value must be a value greater than or equal to zero, or the following special value which is recognized:

MQVS_USE_VSLENGTH

When specified, the length of the buffer is taken from the VSLength field in the MQCHARV structure. Do not use this value when using the structure as an output field and a buffer is provided.


This is the initial value of this field.

VSCCSID (MQLONG)

This is the character set identifier of the variable length string addressed by the **VSPtr** or **VSOOffset** field.

The initial value of this field is *MQCCSI_APPL*, which is defined by MQ to indicate that it should be changed to the true character set identifier of the current process. As a result, the value of constant *MQCCSI_APPL* is never associated with a variable length string.

The initial value of this field can be changed by defining a different value for the constant *MQCCSI_APPL* for your compile unit. How you do this depends on your application's programming language.

 On z/OS systems, the default application CCSID used by *MQCCSI_APPL* is defined as follows:

- For batch LE applications using the DLL interface, the default is the CODESET associated with the current locale at the time **MQCONN** is issued (default value is 1047).
- For batch LE applications bound with one of the batch MQ stubs, the default is the CODESET associated with the current locale at the time of the first MQI call issued after **MQCONN** (default value is 1047).
- For batch non-LE applications running on a USS thread, the default is the value of THLICCSID at the time of the first MQI call issued after **MQCONN** (default value is 1047).
- For other batch applications, the default is the CCSID of the queue manager.

VSLength (MQLONG)

The length in bytes of the variable length string addressed by the VSPtr or VSOOffset field.

The initial value of this field is 0. The value must be either greater than or equal to zero or the following special value which is recognized:

MQVS_NULL_TERMINATED

If MQVS_NULL_TERMINATED is not specified, VSLength bytes are included as part of the string. If null characters are present they do not delimit the string.

If MQVS_NULL_TERMINATED is specified, the string is delimited by the first null encountered in the string. The null itself is not included as part of that string.

Note: The null character used to terminate a string if MQVS_NULL_TERMINATED is specified is a null from the codeset specified by VSCCSID.

For example, in UTF-16 (UCS-2 CCSIDs 1200 and 13488), this is the two byte Unicode encoding where a null is represented by a 16-bit number of all zeros. In UTF-16 it is common to find single bytes set to all zero which are part of characters (7-bit ASCII characters for instance), but the strings will only be null terminated when two 'zero' bytes are found on an even byte boundary. It is possible to get two 'zero' bytes on an odd boundary when they are each part of valid characters. For example x'01' x'00 x'00' x'30' represents two valid Unicode characters and does not null terminate the string.

VSOOffset (MQLONG)

The offset can be positive or negative. You can use either the VSPtr or VSOOffset field to specify the variable length string, but not both. The offset in bytes of the variable length string from the start of the MQCHARV, or the structure containing it.

When the MQCHARV structure is embedded within another structure, this value is the offset in bytes of the variable length string from the start of the structure that contains this MQCHARV structure. When the MQCHARV structure is not embedded within another structure, for example, if it is specified as a parameter on a function call, the offset is relative to the start of the MQCHARV structure.

The initial value of this field is 0.

VSPtr (MQPTR)

This is a pointer to the variable length string.

You can use either the VSPtr or VSOOffset field to specify the variable length string, but not both.

The initial value of this field is a null pointer or null bytes.

Initial values and language declarations for MQCHARV

Initial values of fields in MQCHARV

Field name	Name of constant	Value of constant
VSPtr	None	Null pointer or null bytes.
VSOOffset	None	0
VBufSize	MQVS_USE_VSLENGTH	0
VSLength	None	0
VSCCSID	MQCCSI_APPL	-3

Note: In the C programming language, the macro variable MQCHARV_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure:

```
MQCHARV MyVarStr = {MQCHARV_DEFAULT};
```

C declaration

```
typedef struct tagMQCHARV MQCHARV;  
struct tagMQCHARV {
```

```

MQPTR      VSPtr;                /* Address of variable length string */
MQLONG     VSOFFSET;             /* Offset of variable length string */
MQLONG     VBufSize;             /* Size of buffer */
MQLONG     VSLength;             /* Length of variable length string */
MQLONG     VSCCSID;              /* CCSID of variable length string */
};

```

COBOL declaration for MQCHARV

```

** MQCHARV structure
10 MQCHARV.
** Address of variable length string
15 MQCHARV-VSPTR      POINTER.
** Offset of variable length string
15 MQCHARV-VSOFFSET   PIC S9(9) BINARY.
** Size of buffer
15 MQCHARV-VBUFSIZE   PIC S9(9) BINARY.
** Length of variable length string
15 MQCHARV-VSLNGTH    PIC S9(9) BINARY.
** CCSID of variable length string
15 MQCHARV-VSCCSID    PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQCHARV based,
3 VSPtr      pointer,          /* Address of variable length string */
3 VSOFFset   fixed bin(31),    /* Offset of variable length string */
3 VBufSize   fixed bin(31),    /* Size of buffer */
3 VSLength   fixed bin(31),    /* Length of variable length string */
3 VSCCSID    fixed bin(31);    /* CCSID of variable length string */

```

High Level Assembler declaration

```

MQCHARV      DSECT
MQCHARV_VSPTR DS F      Address of variable length string
MQCHARV_VSOFFSET DS F    Offset of variable length string
MQCHARV_VBUFSIZE DS F    Size of buffer
MQCHARV_VSLNGTH DS F     Length of variable length string
MQCHARV_VSCCSID DS F     CCSID of variable length string
*
MQCHARV_LENGTH EQU *-MQCHARV
ORG MQCHARV
MQCHARV_AREA DS CL(MQCHARV_LENGTH)

```

Redefinition of MQCCSI_APPL

The following examples show how you can override the value of MQCCSI_APPL in various programming languages. You can change the value of MQCCSI_APPL, removing the need to set the VSCCSID for each variable length string separately.

In these examples the CCSID is set to 1208; change this to the value you require. This becomes the default value, which you can override by setting the VSCCSID in any specific instance of MQCHARV.

C usage

```

#define MQCCSI_APPL 1208
#include <cmqc.h>

```

COBOL usage

```

COPY CMQXYZV REPLACING -3 BY 1208.

```

PL/I usage

```

%MQCCSI_APPL = '1208';
#include syslib(cmqp);

```

System/390 assembler usage

MQCCSI APPL EQU 1208
CMQA LIST=NO

MQCIH - CICS bridge header

The following table summarizes the fields in the structure.

Table 471. Fields in MQCIH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQCIH structure	StrucLength
<i>Encoding</i>	Reserved	Encoding
<i>CodedCharSetId</i>	Reserved	CodedCharSetId
<i>Format</i>	MQ format name of data that follows MQCIH	Format
<i>Flags</i>	Flags	Flags
<i>ReturnCode</i>	Return code from bridge	ReturnCode
<i>CompCode</i>	MQ completion code or CICS EIBRESP	CompCode
<i>Reason</i>	MQ reason or feedback code, or CICS EIBRESP2	Reason
<i>UOWControl</i>	Unit-of-work control	UOWControl
<i>GetWaitInterval</i>	Wait interval for MQGET call issued by bridge task	GetWaitInterval
<i>LinkType</i>	Link type	LinkType
<i>OutputDataLength</i>	Output COMMAREA data length	OutputDataLength
<i>FacilityKeepTime</i>	Bridge facility release time	FacilityKeepTime
<i>ADSDescriptor</i>	Send/receive ADS descriptor	ADSDescriptor
<i>ConversationalTask</i>	Whether task can be conversational	ConversationalTask
<i>TaskEndStatus</i>	Status at end of task	TaskEndStatus
<i>Facility</i>	Bridge facility token	Facility
<i>Function</i>	MQ call name or CICS EIBFN function	Function
<i>AbendCode</i>	Abend code	AbendCode
<i>Authenticator</i>	Password or passticket	Authenticator
<i>Reserved1</i>	Reserved	Reserved1
<i>ReplyToFormat</i>	MQ format name of reply message	ReplyToFormat
<i>RemoteSysId</i>	Remote CICS system Id to use	RemoteSysId
<i>RemoteTransId</i>	CICS RTRANSID to use	RemoteTransId
<i>TransactionId</i>	Transaction to attach	TransactionId

Table 471. Fields in MQCIH (continued)		
Field	Description	Topic
<i>FacilityLike</i>	Terminal emulated attributes	FacilityLike
<i>AttentionId</i>	AID key	AttentionId
<i>StartCode</i>	Transaction start code	StartCode
<i>CancelCode</i>	Abend transaction code	CancelCode
<i>NextTransactionId</i>	Next transaction to attach	NextTransactionId
<i>Reserved2</i>	Reserved	Reserved2
<i>Reserved3</i>	Reserved	Reserved3
Note: The remaining fields are not present if <i>Version</i> is less than MQCIH_VERSION_2.		
<i>CursorPosition</i>	Cursor position	CursorPosition
<i>ErrorOffset</i>	Offset of error in message	ErrorOffset
<i>InputItem</i>	Reserved	InputItem
<i>Reserved4</i>	Reserved	Reserved4

Overview for MQCIH

The MQCIH structure describes the header information for a message sent to CICS across the CICS bridge. For any IBM MQ supported platform you can create and transmit a message that includes the MQCIH structure, but only an IBM MQ for z/OS queue manager can use the CICS bridge. Therefore, for the message to get to CICS from a non-z/OS queue manager, your queue manager network must include at least one z/OS queue manager through which the message can be routed.

Availability: AIX, HP-UX, z/OS, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQCIH structure describes the information that can be present at the start of a message sent to the CICS bridge through IBM MQ for z/OS.

Format name: MQFMT_CICS.

Version: The current version of MQCIH is MQCIH_VERSION_2. Fields that exist only in the more-recent version of the structure are identified as such in the descriptions that follow.

The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQCIH, with the initial value of the *Version* field set to MQCIH_VERSION_2.

Character set and encoding: Special conditions apply to the character set and encoding used for the MQCIH structure and application message data:

- Applications that connect to the queue manager that owns the CICS bridge queue must provide an MQCIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQCIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQCIH structure that is in any of the supported character sets and encodings; the receiving message channel agent connected to the queue manager that owns the CICS bridge queue converts the MQCIH structure.
- The application message data following the MQCIH structure must be in the same character set and encoding as the MQCIH structure. You cannot use the *CodedCharSetId* and *Encoding* fields in the MQCIH structure to specify the character set and encoding of the application message data.

You must provide a data-conversion exit to convert the application message data if the data is not one of the built-in formats supported by the queue manager.

Usage: If the application requires values that are the same as the initial values shown in [Table 473 on page 298](#), and the bridge is running with AUTH=LOCAL or AUTH=IDENTIFY, you can omit the MQCIH structure from the message. In all other cases, the structure must be present.

The bridge accepts either a version-1 or a version-2 MQCIH structure, but for 3270 transactions, you must use a version-2 structure.

The application must ensure that fields documented as request fields have appropriate values in the message sent to the bridge; these fields are input to the bridge.

Fields documented as response fields are set by the CICS bridge in the reply message that the bridge sends to the application. Error information is returned in the *ReturnCode*, *Function*, *CompCode*, *Reason*, and *AbendCode* fields, but not all of them are set in all cases. [Table 472 on page 290](#) shows which fields are set for different values of *ReturnCode*.

<i>Table 472. Contents of error information fields in MQCIH structure for MQCIH</i>				
ReturnCode	Function	CompCode	Reason	AbendCode
MQCRC_OK	-	-	-	-
MQCRC_BRIDGE_ERROR	-	-	MQFB_CICS_*	-
MQCRC_MQ_API_ERROR MQCRC_BRIDGE_TIMEOUT	MQ call name	MQ CompCode	MQ Reason	-
MQCRC_CICS_EXEC_ERROR MQCRC_SECURITY_ERROR MQCRC_PROGRAM_NOT_AVAILABLE MQCRC_TRANSID_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	-
MQCRC_BRIDGE_ABEND MQCRC_APPLICATION_ABEND	-	-	-	CICS ABCODE

Fields for MQCIH

The MQCIH structure contains the following fields; the fields are described in **alphabetical order**:

AbendCode (MQCHAR4)

AbendCode is a response field. The length of this field is given by MQ_ABEND_CODE_LENGTH. The initial value of this field is 4 blank characters.

The value returned in this field is significant only if the *ReturnCode* field has the value MQCRC_APPLICATION_ABEND or MQCRC_BRIDGE_ABEND. If it does, *AbendCode* contains the CICS ABCODE value.

ADSDescriptor (MQLONG)

This field is an indicator specifying whether to send ADS descriptors on SEND and RECEIVE BMS requests.

The following values are defined:

MQCADSD_NONE

Do not send or receive ADS descriptors.

MQCADSD_SEND

Send ADS descriptors.

MQCADSD_RECV

Receive ADS descriptors.

MQCADSD_MSGFORMAT

Use message format for the ADS descriptors.

This sends or receives the ADS descriptors using the long form of the ADS descriptor. The long form has fields that are aligned on 4-byte boundaries.

Set the *ADSDescriptor* field as follows:

- If you are not using ADS descriptors, set the field to MQCADSD_NONE.
- If you are using ADS descriptors with the *same* CCSID in each environment, set the field to the sum of MQCADSD_SEND and MQCADSD_RECV.
- If you are using ADS descriptors with *different* CCSIDs in each environment, set the field to the sum of MQCADSD_SEND, MQCADSD_RECV, and MQCADSD_MSGFORMAT.

This is a request field used only for 3270 transactions. The initial value of this field is MQCADSD_NONE.

AttentionId (MQCHAR4)

The value in this field determines the initial value of the AID key when the transaction is started. It is a 1 byte value, left-aligned.

AttentionId is a request field used only for 3270 transactions. The length of this field is given by MQ_ATTENTION_ID_LENGTH. The initial value of this field is four blanks.

Authenticator (MQCHAR8)

The value of this field is the password or passticket.

If user-identifier authentication is active for the CICS bridge, *Authenticator* is used with the user identifier in the MQMD identity context to authenticate the sender of the message.

This is a request field. The length of this field is given by MQ_AUTHENTICATOR_LENGTH. The initial value of this field is 8 blanks.

CancelCode (MQCHAR4)

The value in this field is the abend code to be used to terminate the transaction (normally a conversational transaction that is requesting more data). Otherwise this field is set to blanks.

This field is a request field used only for 3270 transactions. The length of this field is given by MQ_CANCEL_CODE_LENGTH. The initial value of this field is four blanks.

CodedCharSetId (MQLONG)

CodedCharSetId is a reserved field; its value is not significant. The initial value of this field is 0.

The Character Set ID for supported structures which follow an MQCIH structure is the same as the Character Set ID of the MQCIH structure itself and is taken from any preceding IBM MQ header.

CompCode (MQLONG)

This field is a response field. Its initial value is MQCC_OK

The value returned in this field depends on *ReturnCode* ; see [Table 472 on page 290](#).

ConversationalTask (MQLONG)

This field is an indicator specifying whether to allow the task to issue requests for more information, or to stop the task and issue an abend message.

The value must be one of the following options:

MQCCT_YES

The task is conversational.

MQCCT_NO

The task is not conversational.

This field is a request field used only for 3270 transactions. The initial value of this field is MQCCT_NO.

CursorPosition (MQLONG)

The value in this field shows the initial cursor position when the transaction is started. For conversational transactions, the cursor position is in the RECEIVE vector.

This field is a request field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *Version* is less than MQCIH_VERSION_2.

Encoding (MQLONG)

This field is a reserved field; its value is not significant. Its initial value is 0.

The Encoding for supported structures which follow an MQCIH structure is the same as the Encoding of the MQCIH structure itself and taken from any preceding IBM MQ header.

ErrorOffset (MQLONG)

The ErrorOffset field shows the position of invalid data detected by the bridge exit. This field provides the offset from the start of the message to the location of the invalid data.

ErrorOffset is a response field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *Version* is less than MQCIH_VERSION_2.

Facility (MQBYTE8)

This field shows the 8-byte bridge facility token.

A bridge facility token enables multiple transactions in a pseudo-conversation to use the same bridge facility (virtual 3270 terminal). In the first, or only, message in a pseudo-conversation, set a value of MQCFAC_NONE. This value tells CICS to allocate a new bridge facility for this message. A bridge facility token is returned in response messages when a nonzero *FacilityKeepTime* is specified on the input message. Subsequent input messages within a pseudo-conversation must then use the same bridge facility token.

The following special value is defined:

MQCFAC_NONE

No facility token specified.

For the C programming language, the constant MQCFAC_NONE_ARRAY is also defined, and has the same value as MQCFAC_NONE, but is an array of characters instead of a string.

This field is both a request and a response field used only for 3270 transactions. The length of this field is given by MQ_FACILITY_LENGTH. The initial value of this field is MQCFAC_NONE.

FacilityKeepTime (MQLONG)

FacilityKeepTime is the length of time in seconds that the bridge facility is kept after the user transaction ends.

For pseudo-conversational transactions, specify a value that corresponds to the expected duration of a pseudo-conversation; specify zero for the last transaction of a pseudo-conversation, and for other transaction types specify zero.

This field is a request field used only for 3270 transactions. The initial value of this field is 0.

FacilityLike (MQCHAR4)

FacilityLike is the name of an installed terminal that is to be used as a model for the bridge facility.

A value of blanks means that *FacilityLike* is taken from the bridge transaction profile definition, or a default value is used.

This field is a request field used only for 3270 transactions. The length of this field is given by MQ_FACILITY_LIKE_LENGTH. The initial value of this field is four blanks.

Flags (MQLONG)

This field is a request field. The initial value of this field is MQCIH_NONE.

The value must be:

MQCIH_NONE

No flags.

MQCIH_PASS_EXPIRATION

The reply message contains:

- The same expiry report options as the request message.

- The remaining expiry time from the request message with no adjustment made for the processing time of the bridge.

If you omit this value, the expiry time is set to *unlimited*.

MQCIH_REPLY_WITHOUT_NULLS

The reply message length of a CICS DPL program request is adjusted to exclude trailing nulls (X'00') at the end of the COMMAREA returned by the DPL program. If this value is not set, the nulls might be significant, and the full COMMAREA is returned.

MQCIH_SYNC_ON_RETURN

The CICS link for DPL requests uses the SYNCONRETURN option, causing CICS to take a sync point when the program completes if it is shipped to another CICS region. The bridge does not specify to which CICS region to ship the request; that is controlled by the CICS program definition or workload balancing facilities.

Format (MQCHAR8)

This field shows the IBM MQ format name of the data that follows the MQCIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as the rules for coding the *Format* field in MQMD.

This format name is also used for the reply message, if the *ReplyToFormat* field has the value MQFMT_NONE.

- For DPL requests, *Format* must be the format name of the COMMAREA.
- For 3270 requests, *Format* must be CSQCBDCI, and the bridge sets the format to CSQCBDC0 for Reply messages.

The data-conversion exits for these formats must be installed on the queue manager where they are to run.

If the request message generates an error reply message, the error reply message has a format name of MQFMT_STRING.

This field is a request field. The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Function (MQCHAR4)

This field is a response field. The length of this field is given by MQ_FUNCTION_LENGTH. The initial value of this field is MQCFUNC_NONE.

The value returned in this field depends on *ReturnCode* ; see [Table 472 on page 290](#). The following values are possible when *Function* contains an IBM MQ call name:

MQCFUNC_MQCONN

MQCONN call.

MQCFUNC_MQGET

MQGET call.

MQCFUNC_MQINQ

MQINQ call.

MQCFUNC_MQOPEN

MQOPEN call.

MQCFUNC_MQPUT

MQPUT call.

MQCFUNC_MQPUT1

MQPUT1 call.

MQCFUNC_NONE

No call.

In all cases, for the C programming language the constants MQCFUNC_*_ARRAY are also defined; these constants have the same values as the corresponding MQCFUNC_* constants, but are arrays of characters instead of strings.

GetWaitInterval (MQLONG)

This field is a request field. Its initial value is MQCGWI_DEFAULT.

This field applies only when *UOWControl* has the value MQCUOWC_FIRST. It enables the sending application to specify the approximate time in milliseconds that the MQGET calls issued by the bridge will wait for second and subsequent request messages for the unit of work started by this message. This facility overrides the default wait interval used by the bridge. You can use the following special values:

MQCGWI_DEFAULT

Default wait interval.

This value causes the CICS bridge to wait for the time specified when the bridge was started.

MQWI_UNLIMITED

Unlimited wait interval.

InputItem (MQLONG)

This field is a reserved field. The value must be 0.

This field is not present if *Version* is less than MQCIH_VERSION_2.

LinkType (MQLONG)

This field is a request field. Its initial value is MQCLT_PROGRAM.

This value indicates the type of object that the bridge tries to link. It must be one of the following values:

MQCLT_PROGRAM

DPL program.

MQCLT_TRANSACTION

3270 transaction.

NextTransactionId (MQCHAR4)

This value is the name of the next transaction returned by the user transaction (usually by EXEC CICS RETURN TRANSID). If there is no next transaction, this field is set to blanks.

This field is a response field used only for 3270 transactions. The length of this field is given by MQ_TRANSACTION_ID_LENGTH. The initial value of this field is four blanks.

OutputDataLength (MQLONG)

This field is a request field used only for DPL programs. Its initial value is MQCODL_AS_INPUT.

This value is the length of the user data to be returned to the client in a reply message. This length includes the 8-byte program name. The length of the COMMAREA passed to the linked program is the maximum of this field and the length of the user data in the request message, minus 8.

Note: The length of the user data in a message is the length of the message excluding the MQCIH structure.

If the length of the user data in the request message is smaller than *OutputDataLength*, the DATALENGTH option of the LINK command is used, enabling the LINK to be function-shipped efficiently to another CICS region.

You can use the following special value:

MQCODL_AS_INPUT

Output length is same as input length.

This value might be needed even if no reply is requested, in order to ensure that the COMMAREA passed to the linked program is of sufficient size.

Reason (MQLONG)

This field is a response field. Its initial value is MQRC_NONE.

The value returned in this field depends on *ReturnCode* ; see [Table 472 on page 290](#).

RemoteSysId (MQCHAR4)

This field shows the CICS system identifier of the CICS system processing the request.

If this field is blank, the CICS system request is processed on the same CICS system as the bridge monitor. The SYSID used is returned in the Reply message.

For a 3270 pseudo-conversation, all subsequent messages in the conversation must specify the remote SYSID returned in the initial reply. If specified, the SYSID must:

- Be active.
- Have access to the IBM MQ Request queue.
- Be accessible by the CICS ISC links from the CICS system of the bridge monitor.

RemoteTransId (MQCHAR4)

This field is an optional Request field. The length of this field is given by MQ_TRANSACTION_ID_LENGTH.

If specified, the field is used as the RTRANSID value of CICS START.

ReplyToFormat (MQCHAR8)

The value of this field is the IBM MQ format name of the reply message that is sent in response to the current message.

The rules for coding this field are the same as those rules for coding the *Format* field in MQMD.

This field is a request field used only for DPL programs. The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Reserved1 (MQCHAR8)

This field is a reserved field. The value must be 8 blanks.

Reserved2 (MQCHAR8)

This field is a reserved field. The value must be 8 blanks.

Reserved3 (MQCHAR8)

This field is a reserved field. The value must be 8 blanks.

Reserved4 (MQLONG)

This field is a reserved field. The value must be 0.

This field is not present if *Version* is less than MQCIH_VERSION_2.

ReturnCode (MQLONG)

The value of this field is the return code from the CICS bridge describing the outcome of the processing performed by the bridge. This field is a response field, with an initial value of MQCRC_OK.

The *Function*, *CompCode*, *Reason*, and *AbendCode* fields might contain additional information (see [Table 472 on page 290](#)). The value is one of the following:

MQCRC_APPLICATION_ABEND

(5, X'005') Application ended abnormally.

MQCRC_BRIDGE_ABEND

(4, X'004') CICS bridge ended abnormally.

MQCRC_BRIDGE_ERROR

(3, X'003') CICS bridge detected an error.

MQCRC_BRIDGE_TIMEOUT

(8, X'008') Second or later message within current unit of work not received within specified time.

MQCRC_CICS_EXEC_ERROR

(1, X'001') EXEC CICS statement detected an error.

MQCRC_MQ_API_ERROR

(2, X'002') MQ call detected an error.

MQCRC_OK

(0, X'000') No error.

MQCRC_PROGRAM_NOT_AVAILABLE

(7, X'007') Program not available.

MQCRC_SECURITY_ERROR

(6, X'006') Security error occurred.

MQCRC_TRANSID_NOT_AVAILABLE

(9, X'009') Transaction not available.

StartCode (MQCHAR4)

The value of this field is an indicator specifying whether the bridge emulates a terminal transaction or a transaction initiated with START.

The value must be one of the following:

MQCSC_START

Start.

MQCSC_STARTDATA

Start data.

MQCSC_TERMINPUT

Terminal input.

MQCSC_NONE

None.

In all cases, for the C programming language the constants MQCSC_*_ARRAY are also defined; these constants have the same values as the corresponding MQCSC_* constants, but are arrays of characters instead of strings.

In the response from the bridge, this field is set to the start code appropriate to the next transaction ID contained in the *NextTransactionId* field. The following start codes are possible in the response:

- MQCSC_START
- MQCSC_STARTDATA
- MQCSC_TERMINPUT

For CICS Transaction Server Version 1.2, this field is a request field only; its value in the response is undefined.

For CICS Transaction Server Version 1.3 and subsequent releases, this field is both a request and a response field.

This field is used only for 3270 transactions. The length of this field is given by MQ_START_CODE_LENGTH. The initial value of this field is MQCSC_NONE.

StrucId (MQCHAR4)

This field is a request field, with an initial value of MQCIH_STRUC_ID.

The value must be:

MQCIH_STRUC_ID

Identifier for CICS information header structure.

For the C programming language, the constant MQCIH_STRUC_ID_ARRAY is also defined; this has the same value as MQCIH_STRUC_ID, but is an array of characters instead of a string.

StrucLength (MQLONG)

This field is a request field, with an initial value of MQCIH_LENGTH_2.

The value must be one of the following:

MQCIH_LENGTH_1

Length of version-1 CICS information header structure.

MQCIH_LENGTH_2

Length of version-2 CICS information header structure.

The following constant specifies the length of the current version:

MQCIH_CURRENT_LENGTH

Length of current version of CICS information header structure.

TaskEndStatus (MQLONG)

This field is a response field, showing the status of the user transaction at end of task. The field is used only for 3270 transactions, and its initial value is MQCTES_NOSYNC.

One of the following values is returned:

MQCTES_NOSYNC

Not synchronized.

The user transaction has not yet completed and has not syncpointed. The *MsgType* field in MQMD is MQMT_REQUEST in this case.

MQCTES_COMMIT

Commit unit of work.

The user transaction has not yet completed, but has syncpointed the first unit of work. The *MsgType* field in MQMD is MQMT_DATAGRAM in this case.

MQCTES_BACKOUT

Back out unit of work.

The user transaction has not yet completed. The current unit of work is backed out. The *MsgType* field in MQMD is MQMT_DATAGRAM in this case.

MQCTES_ENDTASK

End task.

The user transaction has ended (or abended). The *MsgType* field in MQMD is MQMT_REPLY in this case.

TransactionId (MQCHAR4)

This field is a request field. Its length is given by MQ_TRANSACTION_ID_LENGTH. The initial value of this field is four blanks.

If *LinkType* has the value MQCLT_TRANSACTION, *TransactionId* is the transaction identifier of the user transaction to be run; specify a nonblank value in this case.

If *LinkType* has the value MQCLT_PROGRAM, *TransactionId* is the transaction code under which all programs within the unit of work are to be run. If you specify a blank value, the CICS DPL bridge default transaction code (CKBP) is used. If the value is nonblank, you must have defined it to CICS as a local transaction with an initial program that is CSQCBP00. This field applies only when *UOWControl* has the value MQCUOWC_FIRST or MQCUOWC_ONLY.

UOWControl (MQLONG)

This field is a request field which controls the unit-of-work processing performed by the CICS bridge. The initial value of this field is MQCUOWC_ONLY.

You can request the bridge to run a single transaction, or one or more programs within a unit of work. The field indicates whether the CICS bridge starts a unit of work, performs the requested function within the current unit of work, or ends the unit of work by committing it or backing it out. Various combinations are supported, to optimize the data transmission flows.

The value must be one of the following:

MQCUOWC_ONLY

Start unit of work, perform function, then commit the unit of work.

MQCUOWC_CONTINUE

Additional data for the current unit of work (3270 only).

MQCUOWC_FIRST

Start unit of work and perform function.

MQCUOWC_MIDDLE

Perform function within current unit of work

MQCUOWC_LAST

Perform function, then commit the unit of work.

MQCUOWC_COMMIT

Commit the unit of work (DPL only).

MQCUOWC_BACKOUT

Back out the unit of work (DPL only).

Version (MQLONG)

This field is a request field. Its initial value is MQCIH_VERSION_2.

The value must be one of the following:

MQCIH_VERSION_1

Version-1 CICS information header structure.

MQCIH_VERSION_2

Version-2 CICS information header structure.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQCIH_CURRENT_VERSION

Current version of CICS information header structure.

Initial values and language declarations for MQCIH

<i>Table 473. Initial values of fields in MQCIH for MQCIH</i>		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCIH_STRUC_ID	'CIH '
<i>Version</i>	MQCIH_VERSION_2	2
<i>StrucLength</i>	MQCIH_LENGTH_2	180
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	None	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQCIH_NONE	0
<i>ReturnCode</i>	MQCRC_OK	0

Table 473. Initial values of fields in MQCIH for MQCIH (continued)

Field name	Name of constant	Value of constant
<i>CompCode</i>	MQCC_OK	0
<i>Reason</i>	MQRC_NONE	0
<i>UOWControl</i>	MQCUOWC_ONLY	273
<i>GetWaitInterval</i>	MQCGWI_DEFAULT	-2
<i>LinkType</i>	MQCLT_PROGRAM	1
<i>OutputDataLength</i>	MQCODL_AS_INPUT	-1
<i>FacilityKeepTime</i>	None	0
<i>ADSDescriptor</i>	MQCADSD_NONE	0
<i>ConversationalTask</i>	MQCCT_NO	0
<i>TaskEndStatus</i>	MQCTES_NOSYNC	0
<i>Facility</i>	MQCFAC_NONE	Nulls
<i>Function</i>	MQCFUNC_NONE	Blanks
<i>AbendCode</i>	None	Blanks
<i>Authenticator</i>	None	Blanks
<i>Reserved1</i>	None	Blanks
<i>ReplyToFormat</i>	MQFMT_NONE	Blanks
<i>RemoteSysId</i>	None	Blanks
<i>RemoteTransId</i>	None	Blanks
<i>TransactionId</i>	None	Blanks
<i>FacilityLike</i>	None	Blanks
<i>AttentionId</i>	None	Blanks
<i>StartCode</i>	MQCSC_NONE	Blanks
<i>CancelCode</i>	None	Blanks
<i>NextTransactionId</i>	None	Blanks
<i>Reserved2</i>	None	Blanks
<i>Reserved3</i>	None	Blanks
<i>CursorPosition</i>	None	0
<i>ErrorOffset</i>	None	0
<i>InputItem</i>	None	0
<i>Reserved4</i>	None	0

Table 473. Initial values of fields in MQCIH for MQCIH (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. In the C programming language, the macro variable MQCIH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQCIH MyCIH = {MQCIH_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQCIH MQCIH;
struct tagMQCIH {
    MQCHAR4  StrucId;           /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   StrucLength;      /* Length of MQCIH structure */
    MQLONG   Encoding;         /* Reserved */
    MQLONG   CodedCharSetId;   /* Reserved */
    MQCHAR8  Format;           /* MQ format name of data that follows
                               MQCIH */
    MQLONG   Flags;            /* Flags */
    MQLONG   ReturnCode;       /* Return code from bridge */
    MQLONG   CompCode;         /* MQ completion code or CICS EIBRESP */
    MQLONG   Reason;           /* MQ reason or feedback code, or CICS
                               EIBRESP2 */
    MQLONG   UOWControl;       /* Unit-of-work control */
    MQLONG   GetWaitInterval;  /* Wait interval for MQGET call issued
                               by bridge task */
    MQLONG   LinkType;         /* Link type */
    MQLONG   OutputDataLength; /* Output COMMAREA data length */
    MQLONG   FacilityKeepTime; /* Bridge facility release time */
    MQLONG   ADSDescriptor;    /* Send/receive ADS descriptor */
    MQLONG   ConversationalTask; /* Whether task can be conversational */
    MQLONG   TaskEndStatus;    /* Status at end of task */
    MQBYTE8  Facility;         /* Bridge facility token */
    MQCHAR4  Function;         /* MQ call name or CICS EIBFN
                               function */
    MQCHAR4  AbendCode;        /* Abend code */
    MQCHAR8  Authenticator;    /* Password or passticket */
    MQCHAR8  Reserved1;        /* Reserved */
    MQCHAR8  ReplyToFormat;    /* MQ format name of reply message */
    MQCHAR4  RemoteSysId;      /* Reserved */
    MQCHAR4  RemoteTransId;    /* Reserved */
    MQCHAR4  TransactionId;    /* Transaction to attach */
    MQCHAR4  FacilityLike;     /* Terminal emulated attributes */
    MQCHAR4  AttentionId;      /* AID key */
    MQCHAR4  StartCode;        /* Transaction start code */
    MQCHAR4  CancelCode;       /* Abend transaction code */
    MQCHAR4  NextTransactionId; /* Next transaction to attach */
    MQCHAR8  Reserved2;        /* Reserved */
    MQCHAR8  Reserved3;        /* Reserved */
    MQLONG   CursorPosition;   /* Cursor position */
    MQLONG   ErrorOffset;      /* Offset of error in message */
    MQLONG   InputItem;        /* Reserved */
    MQLONG   Reserved4;        /* Reserved */
};
```

COBOL declaration

```
** MQCIH structure
10 MQCIH.
**   Structure identifier
15 MQCIH-STRUCID          PIC X(4).
**   Structure version number
15 MQCIH-VERSION         PIC S9(9) BINARY.
**   Length of MQCIH structure
15 MQCIH-STRUCLNGTH      PIC S9(9) BINARY.
**   Reserved
15 MQCIH-ENCODING        PIC S9(9) BINARY.
**   Reserved
15 MQCIH-CODEDCHARSETID  PIC S9(9) BINARY.
```

```

**      MQ format name of data that follows MQCIH
15      MQCIH-FORMAT          PIC X(8).
**      Flags
15      MQCIH-FLAGS          PIC S9(9) BINARY.
**      Return code from bridge
15      MQCIH-RETURNCODE      PIC S9(9) BINARY.
**      MQ completion code or CICS EIBRESP
15      MQCIH-COMPCODE        PIC S9(9) BINARY.
**      MQ reason or feedback code, or CICS EIBRESP2
15      MQCIH-REASON          PIC S9(9) BINARY.
**      Unit-of-work control
15      MQCIH-UOWCONTROL      PIC S9(9) BINARY.
**      Wait interval for MQGET call issued by bridge task
15      MQCIH-GETWAITINTERVAL PIC S9(9) BINARY.
**      Link type
15      MQCIH-LINKTYPE        PIC S9(9) BINARY.
**      Output COMMAREA data length
15      MQCIH-OUTPUTDATALENGTH PIC S9(9) BINARY.
**      Bridge facility release time
15      MQCIH-FACILITYKEEPTIME PIC S9(9) BINARY.
**      Send/receive ADS descriptor
15      MQCIH-ADSDESCRIPTOR   PIC S9(9) BINARY.
**      Whether task can be conversational
15      MQCIH-CONVERSATIONALTASK PIC S9(9) BINARY.
**      Status at end of task
15      MQCIH-TASKENDSTATUS    PIC S9(9) BINARY.
**      Bridge facility token
15      MQCIH-FACILITY        PIC X(8).
**      MQ call name or CICS EIBFN function
15      MQCIH-FUNCTION        PIC X(4).
**      Abend code
15      MQCIH-ABENDCODE       PIC X(4).
**      Password or passticket
15      MQCIH-AUTHENTICATOR    PIC X(8).
**      Reserved
15      MQCIH-RESERVED1        PIC X(8).
**      MQ format name of reply message
15      MQCIH-REPLYTOFORMAT    PIC X(8).
**      Reserved
15      MQCIH-REMOTESYSID      PIC X(4).
**      Reserved
15      MQCIH-REMOTETRANSID    PIC X(4).
**      Transaction to attach
15      MQCIH-TRANSACTIONID    PIC X(4).
**      Terminal emulated attributes
15      MQCIH-FACILITYLIKE     PIC X(4).
**      AID key
15      MQCIH-ATTENTIONID      PIC X(4).
**      Transaction start code
15      MQCIH-STARTCODE        PIC X(4).
**      Abend transaction code
15      MQCIH-CANCELCODE       PIC X(4).
**      Next transaction to attach
15      MQCIH-NEXTTRANSACTIONID PIC X(4).
**      Reserved
15      MQCIH-RESERVED2        PIC X(8).
**      Reserved
15      MQCIH-RESERVED3        PIC X(8).
**      Cursor position
15      MQCIH-CURSORPOSITION   PIC S9(9) BINARY.
**      Offset of error in message
15      MQCIH-ERROROFFSET      PIC S9(9) BINARY.
**      Reserved
15      MQCIH-INPUTITEM        PIC S9(9) BINARY.
**      Reserved
15      MQCIH-RESERVED4        PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQCIH based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version number */
    3 StrucLength       fixed bin(31),    /* Length of MQCIH structure */
    3 Encoding          fixed bin(31),    /* Reserved */
    3 CodedCharSetId    fixed bin(31),    /* Reserved */
    3 Format             char(8),          /* MQ format name of data that
                                          follows MQCIH */
    3 Flags             fixed bin(31),    /* Flags */
    3 ReturnCode         fixed bin(31),    /* Return code from bridge */

```

3 CompCode	fixed bin(31), /* MQ completion code or CICS EIBRESP */
3 Reason	fixed bin(31), /* MQ reason or feedback code, or CICS EIBRESP2 */
3 UOWControl	fixed bin(31), /* Unit-of-work control */
3 GetWaitInterval	fixed bin(31), /* Wait interval for MQGET call issued by bridge task */
3 LinkType	fixed bin(31), /* Link type */
3 OutputDataLength	fixed bin(31), /* Output COMMAREA data length */
3 FacilityKeepTime	fixed bin(31), /* Bridge facility release time */
3 ADSDescriptor	fixed bin(31), /* Send/receive ADS descriptor */
3 ConversationalTask	fixed bin(31), /* Whether task can be conversational */
3 TaskEndStatus	fixed bin(31), /* Status at end of task */
3 Facility	char(8), /* Bridge facility token */
3 Function	char(4), /* MQ call name or CICS EIBFN function */
3 AbendCode	char(4), /* Abend code */
3 Authenticator	char(8), /* Password or passticket */
3 Reserved1	char(8), /* Reserved */
3 ReplyToFormat	char(8), /* MQ format name of reply message */
3 RemoteSysId	char(4), /* Reserved */
3 RemoteTransId	char(4), /* Reserved */
3 TransactionId	char(4), /* Transaction to attach */
3 FacilityLike	char(4), /* Terminal emulated attributes */
3 AttentionId	char(4), /* AID key */
3 StartCode	char(4), /* Transaction start code */
3 CancelCode	char(4), /* Abend transaction code */
3 NextTransactionId	char(4), /* Next transaction to attach */
3 Reserved2	char(8), /* Reserved */
3 Reserved3	char(8), /* Reserved */
3 CursorPosition	fixed bin(31), /* Cursor position */
3 ErrorOffset	fixed bin(31), /* Offset of error in message */
3 InputItem	fixed bin(31), /* Reserved */
3 Reserved4	fixed bin(31); /* Reserved */

High Level Assembler declaration

MQCIH	DSECT	
MQCIH_STRUCID	DS	CL4 Structure identifier
MQCIH_VERSION	DS	F Structure version number
MQCIH_STRUCLNGTH	DS	F Length of MQCIH structure
MQCIH_ENCODING	DS	F Reserved
MQCIH_CODEDCHARSETID	DS	F Reserved
MQCIH_FORMAT	DS	CL8 MQ format name of data that follows MQCIH
*		
MQCIH_FLAGS	DS	F Flags
MQCIH_RETURNCODE	DS	F Return code from bridge
MQCIH_COMPCODE	DS	F MQ completion code or CICS EIBRESP
MQCIH_REASON	DS	F MQ reason or feedback code, or CICS EIBRESP2
*		
MQCIH_UOWCONTROL	DS	F Unit-of-work control
MQCIH_GETWAITINTERVAL	DS	F Wait interval for MQGET call issued by bridge task
*		
MQCIH_LINKTYPE	DS	F Link type
MQCIH_OUTPUTDATALENGTH	DS	F Output COMMAREA data length
MQCIH_FACILITYKEEPTIME	DS	F Bridge facility release time
MQCIH_ADSDESCRIPTOR	DS	F Send/receive ADS descriptor
MQCIH_CONVERSATIONALTASK	DS	F Whether task can be conversational
MQCIH_TASKENDSTATUS	DS	F Status at end of task
MQCIH_FACILITY	DS	XL8 Bridge facility token
MQCIH_FUNCTION	DS	CL4 MQ call name or CICS EIBFN function
MQCIH_ABENDCODE	DS	CL4 Abend code
MQCIH_AUTHENTICATOR	DS	CL8 Password or passticket
MQCIH_RESERVED1	DS	CL8 Reserved
MQCIH_REPLYTOFORMAT	DS	CL8 MQ format name of reply message
MQCIH_REMOTESYSID	DS	CL4 Reserved
MQCIH_REMOTETRANSID	DS	CL4 Reserved
MQCIH_TRANSACTIONID	DS	CL4 Transaction to attach
MQCIH_FACILITYLIKE	DS	CL4 Terminal emulated attributes
MQCIH_ATTENTIONID	DS	CL4 AID key
MQCIH_STARTCODE	DS	CL4 Transaction start code
MQCIH_CANCELCODE	DS	CL4 Abend transaction code
MQCIH_NEXTTRANSACTIONID	DS	CL4 Next transaction to attach
MQCIH_RESERVED2	DS	CL8 Reserved
MQCIH_RESERVED3	DS	CL8 Reserved
MQCIH_CURSORPOSITION	DS	F Cursor position
MQCIH_ERROROFFSET	DS	F Offset of error in message
MQCIH_INPUTITEM	DS	F Reserved

MQCIH_RESERVED4	DS	F	Reserved
*MQCIH_LENGTH	EQU	*-MQCIH	
MQCIH_AREA	ORG	MQCIH	
	DS	CL(MQCIH_LENGTH)	

Visual Basic declaration

```

Type MQCIH
    StrucId          As String*4 'Structure identifier'
    Version          As Long      'Structure version number'
    StrucLength      As Long      'Length of MQCIH structure'
    Encoding         As Long      'Reserved'
    CodedCharSetId   As Long      'Reserved'
    Format           As String*8  'MQ format name of data that follows'
                        'MQCIH'
    Flags            As Long      'Flags'
    ReturnCode       As Long      'Return code from bridge'
    CompCode        As Long      'MQ completion code or CICS EIBRESP'
    Reason          As Long      'MQ reason or feedback code, or CICS'
                        'EIBRESP2'
    UOWControl       As Long      'Unit-of-work control'
    GetWaitInterval As Long      'Wait interval for MQGET call issued'
                        'by bridge task'
    LinkType         As Long      'Link type'
    OutputDataLength As Long      'Output COMMAREA data length'
    FacilityKeepTime As Long      'Bridge facility release time'
    ADSDescriptor    As Long      'Send/receive ADS descriptor'
    ConversationalTask As Long    'Whether task can be conversational'
    TaskEndStatus    As Long      'Status at end of task'
    Facility         As MQBYTE8  'Bridge facility token'
    Function         As String*4  'MQ call name or CICS EIBFN function'
    AbendCode        As String*4  'Abend code'
    Authenticator    As String*8  'Password or passticket'
    Reserved1        As String*8  'Reserved'
    ReplyToFormat    As String*8  'MQ format name of reply message'
    RemoteSysId      As String*4  'Reserved'
    RemoteTransId    As String*4  'Reserved'
    TransactionId    As String*4  'Transaction to attach'
    FacilityLike     As String*4  'Terminal emulated attributes'
    AttentionId      As String*4  'AID key'
    StartCode        As String*4  'Transaction start code'
    CancelCode       As String*4  'Abend transaction code'
    NextTransactionId As String*4  'Next transaction to attach'
    Reserved2        As String*8  'Reserved'
    Reserved3        As String*8  'Reserved'
    CursorPosition   As Long      'Cursor position'
    ErrorOffset      As Long      'Offset of error in message'
    InputItem        As Long      'Reserved'
    Reserved4        As Long      'Reserved'
End Type

```

MQCMHO - Create message handle options

The following table summarizes the fields in the structure.

Table 474. Fields in MQCMHO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options

Overview for MQCMHO

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS and IBM MQ clients.

Purpose: The **MQCMHO** structure allows applications to specify options that control how message handles are created. The structure is an input parameter on the **MQCRTMH** call.

Character set and encoding: Data in **MQCMHO** must be in the character set of the application and encoding of the application (**MQENC_NATIVE**).

Fields for MQCMHO

The MQCMHO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

This field is always an input field. Its initial value is MQCMHO_DEFAULT_VALIDATION.

One of the following options can be specified:

MQCMHO_VALIDATE

When **MQSETMP** is called to set a property in this message handle, the property name is validated to ensure that it:

- contains no invalid characters.
- does not begin JMS or usr.JMS except for the following:
 - JMSCorrelationID
 - JMSReplyTo
 - JMSType
 - JMSXGroupID
 - JMSXGroupSeq

These names are reserved for JMS properties.

- is not one of the following keywords, in any mixture of uppercase or lowercase:
 - AND
 - BETWEEN
 - ESCAPE
 - FALSE
 - IN
 - IS
 - LIKE
 - NOT
 - NULL
 - OR
 - TRUE
- does not begin Body. or Root. (except for Root.MQMD.).

If the property is MQ-defined (mq.*) and the name is recognized, the property descriptor fields are set to the correct values for the property. If the property is not recognized, the *Support* field of the property descriptor is set to **MQPD_OPTIONAL**.

MQCMHO_DEFAULT_VALIDATION

This value specifies that the default level of validation of property names occur.

The default level of validation is equivalent to the level specified by **MQCMHO_VALIDATE**.

This value is the default value.

MQCMHO_NO_VALIDATION

No validation on the property name occurs. See the description of **MQCMHO_VALIDATE**.

Default option: If none of the preceding options described is required, the following option can be used:

MQCMHO_NONE

All options assume their default values. Use this value to indicate that no other options have been specified. **MQCMHO_NONE** aids program documentation; it is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

StrucId (MQCHAR4)

This field is always an input field. Its initial value is MQCMHO_STRUC_ID.

This is the structure identifier; the value must be:

MQCMHO_STRUC_ID

Identifier for create message handle options structure.

For the C programming language, the constant **MQCMHO_STRUC_ID_ARRAY** is also defined; this has the same value as **MQCMHO_STRUC_ID**, but is an array of characters instead of a string.

Version (MQLONG)

This field is always an input field. Its initial value is MQCMHO_VERSION_1.

This is the structure version number; the value must be:

MQCMHO_VERSION_1

Version-1 create message handle options structure.

The following constant specifies the version number of the current version:

MQCMHO_CURRENT_VERSION

Current version of create message handle options structure.

Initial values and language declarations for MQCMHO

Table 475. Initial values of fields in MQCMHO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCMHO_STRUC_ID	'CMHO'
<i>Version</i>	MQCMHO_VERSION_1	1
<i>Options</i>	MQCMHO_DEFAULT_VALIDATION	0
Notes: 1. In the C programming language, the macro variable MQCMHO_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQCMHO MyCMHO = {MQCMHO_DEFAULT};</pre>		

C declaration

```
struct tagMQCMHO {  
    MQCHAR4    StrucId;        /* Structure identifier */  
    MQLONG     Version;        /* Structure version number */  
    MQLONG     Options;        /* Options that control the action of MQCRTMH */  
};
```

COBOL declaration

```
** MQCMHO structure
10 MQCMHO.
**   Structure identifier
15 MQCMHO-STRUCID      PIC X(4).
**   Structure version number
15 MQCMHO-VERSION      PIC S9(9) BINARY.
**   Options that control the action of MQCRTMH
15 MQCMHO-OPTIONS      PIC S9(9) BINARY.
```

PL/I declaration

```
dcl
1 MQCMHO based,
3 StrucId      char(4),          /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 Options      fixed bin(31), /* Options that control the action of MQCRTMH */
```

High Level Assembler declaration

```
MQCMHO          DSECT
MQCMHO_STRUCID  DS   CL4      Structure identifier
MQCMHO_VERSION  DS   F        Structure version number
MQCMHO_OPTIONS  DS   F        Options that control the action of
*                               MQCRTMH
MQCMHO_LENGTH   EQU   *-MQCMHO
MQCMHO_AREA     DS   CL(MQCMHO_LENGTH)
```

MQCNO - Connect options

The following table summarizes the fields in the structure.

Table 476. Fields in MQCNO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options that control the action of MQCONN	Options
Note: The remaining fields are ignored if <i>Version</i> is less than MQCNO_VERSION_2.		
<i>ClientConnOffset</i>	Offset of MQCD structure for client connection	ClientConnOffset
<i>ClientConnPtr</i>	Address of MQCD structure for client connection	ClientConnPtr
Note: The remaining fields are ignored if <i>Version</i> is less than MQCNO_VERSION_3.		
<i>ConnTag</i>	Queue-manager connection tag	ConnTag
Note: The remaining fields are ignored if <i>Version</i> is less than MQCNO_VERSION_4.		
<i>SSLConfigPtr</i>	Address of MQSCO structure for client connection	SSLConfigPtr
<i>SSLConfigOffset</i>	Offset of MQSCO structure for client connection	SSLConfigOffset
Note: The remaining fields are ignored if <i>Version</i> is less than MQCNO_VERSION_5.		
<i>ConnectionId</i>	Unique connection ID	ConnectionId

Table 476. Fields in MQCNO (continued)		
Field	Description	Topic
<i>SecurityParmsOffset</i>	Security parameters	SecurityParmsOffset
<i>SecurityParmsPtr</i>	Security parameters	SecurityParmsPtr

Related tasks

[Using MQCONN](#)

Overview for MQCNO

Availability: All versions except MQCNO_VERSION_4: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQCNO structure allows the application to specify options relating to the connection to the local queue manager. The structure is an input/output parameter on the MQCONN call. For more information about using shared handles, and the MQCONN call, see [Shared \(thread independent\) connections with MQCONN](#).

Version: The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQCNO, but with the initial value of the *Version* field set to MQCNO_VERSION_1. To use fields that are not present in the version-1 structure, the application must set the *Version* field to the version number of the version required.

Character set and encoding: Data in MQCNO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an IBM MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQCNO

The MQCNO structure contains the following fields; the fields are described in **alphabetical order**:

ClientConnOffset (MQLONG)

ClientConnOffset is the offset in bytes of an MQCD channel definition structure from the start of the MQCNO structure. The offset can be positive or negative. This field is an input field with an initial value of 0.

Use *ClientConnOffset* only when the application issuing the MQCONN call is running as an IBM MQ MQI client. For information about how to use this field, see the description of the *ClientConnPtr* field.

This field is ignored if *Version* is less than MQCNO_VERSION_2.

ClientConnPtr (MQPTR)

ClientConnPtr is an input field. Its initial value is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Use *ClientConnOffset* and *ClientConnPtr* only when the application issuing the MQCONN call is running as an IBM MQ MQI client. By specifying one or other of these fields, the application can control the definition of the client connection channel by providing an MQCD channel definition structure that contains the values required.

If the application is running as an IBM MQ MQI client, but does not provide an MQCD structure, the MQSERVER environment variable is used to select the channel definition. If MQSERVER is not set, the client channel table is used.

If the application is not running as an IBM MQ MQI client, *ClientConnOffset* and *ClientConnPtr* are ignored.

If the application provides an MQCD structure, set the fields listed to the values required; other fields in MQCD are ignored. You can pad character strings with blanks to the length of the field, or terminated them with a null character. See [“Fields” on page 1488](#) for more information about the fields in the MQCD structure.

Field in MQCD	Value
<i>ChannelName</i>	Channel name.
<i>Version</i>	Structure version number. Must not be less than MQCD_VERSION_7.
<i>TransportType</i>	Any supported transport type.
<i>ModeName</i>	LU 6.2 mode name.
<i>TpName</i>	LU 6.2 transaction program name.
<i>SecurityExit</i>	Name of channel security exit.
<i>SendExit</i>	Name of channel send exit.
<i>ReceiveExit</i>	Name of channel receive exit.
<i>MaxMsgLength</i>	Maximum length in bytes of messages that can be sent over the client connection channel.
<i>SecurityUserData</i>	User data for security exit.
<i>SendUserData</i>	User data for send exit.
<i>ReceiveUserData</i>	User data for receive exit.
<i>UserIdentifier</i>	User identifier to be used to establish an LU 6.2 session.
<i>Password</i>	Password to be used to establish an LU 6.2 session.
<i>ConnectionName</i>	Connection name.
<i>HeartbeatInterval</i>	Time in seconds between heartbeat flows.
<i>StrucLength</i>	Length of the MQCD structure.
<i>ExitNameLength</i>	Length of exit names addressed by <i>SendExitPtr</i> and <i>ReceiveExitPtr</i> . Must be greater than zero if <i>SendExitPtr</i> or <i>ReceiveExitPtr</i> is set to a value that is not the null pointer.
<i>ExitDataLength</i>	Length of exit data addressed by <i>SendUserDataPtr</i> and <i>ReceiveUserDataPtr</i> . Must be greater than zero if <i>SendUserDataPtr</i> or <i>ReceiveUserDataPtr</i> is set to a value that is not the null pointer.
<i>SendExitsDefined</i>	Number of send exits addressed by <i>SendExitPtr</i> . If zero, <i>SendExit</i> and <i>SendUserData</i> provide the exit name and data. If greater than zero, <i>SendExitPtr</i> and <i>SendUserDataPtr</i> provide the exit names and data, and <i>SendExit</i> and <i>SendUserData</i> must be blank.
<i>ReceiveExitsDefined</i>	Number of receive exits addressed by <i>ReceiveExitPtr</i> . If zero, <i>ReceiveExit</i> and <i>ReceiveUserData</i> provide the exit name and data. If greater than zero, <i>ReceiveExitPtr</i> and <i>ReceiveUserDataPtr</i> provide the exit names and data, and <i>ReceiveExit</i> and <i>ReceiveUserData</i> must be blank.
<i>SendExitPtr</i>	Address of name of first send exit.
<i>SendUserDataPtr</i>	Address of data for first send exit.
<i>ReceiveExitPtr</i>	Address of name of first receive exit.
<i>ReceiveUserDataPtr</i>	Address of data for first receive exit.

Field in MQCD	Value
<i>LongRemoteUserIdLength</i>	Length of long remote user identifier.
<i>LongRemoteUserIdPtr</i>	Address of long remote user identifier.
<i>RemoteSecurityId</i>	Remote security identifier.
<i>SSLCipherSpec</i>	SSL CipherSpec.
<i>SSLPeerNamePtr</i>	Address of SSL peer name.
<i>SSLPeerNameLength</i>	Length of SSL peer name.
<i>KeepAliveInterval</i>	Value passed to the communications stack for keepalive timing for the channel
<i>LocalAddress</i>	The local communications address, including the IP address of the local network adapter to use, and a range of ports to use for outgoing connections.

Provide the channel definition structure in one of two ways:

- By using the offset field *ClientConnOffset*

In this case, the application must declare a compound structure containing an MQCNO followed by the channel definition structure MQCD, and set *ClientConnOffset* to the offset of the channel definition structure from the start of the MQCNO. Ensure that this offset is correct. *ClientConnPtr* must be set to the null pointer or null bytes.

Use *ClientConnOffset* for programming languages that do not support the pointer data type, or that implement the pointer data type in a way that is not portable to different environments (for example, the COBOL programming language).

For the Visual Basic programming language, a compound structure called MQCNOCD is provided in the header file CMQXB.BAS; this structure contains an MQCNO structure followed by an MQCD structure. Initialize MQCNOCD by invoking the MQCNOCD_DEFAULTS subroutine. MQCNOCD is used with the MQCONNAny variant of the MQCONN call; see the description of the MQCONN call for further details.

- By using the pointer field *ClientConnPtr*

In this case, the application can declare the channel definition structure separately from the MQCNO structure, and set *ClientConnPtr* to the address of the channel definition structure. Set *ClientConnOffset* to zero.

Use *ClientConnPtr* for programming languages that support the pointer data type in a way that is portable to different environments (for example, the C programming language).

In the C programming language, you can use the macro variable MQCD_CLIENT_CONN_DEFAULT to provide initial values for the structure that are more suitable for use on the MQCONN call than the initial values provided by MQCD_DEFAULT.

Whichever technique you choose, you can use only one of *ClientConnOffset* and *ClientConnPtr*; the call fails with reason code MQRC_CLIENT_CONN_ERROR if both are nonzero.

When the MQCONN call has completed, the MQCD structure is not referenced again.

This field is ignored if *Version* is less than MQCNO_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

ConnectionId (MQBYTE24)

ConnectionId is a unique 24-byte identifier that allows IBM MQ to reliably identify an application. An application can use this identifier for correlation in PUT and GET calls. This output parameter has an initial value of 24 null bytes in all programming languages.

The queue manager assigns a unique ID to all connections, however they are established. If an MQCONN establishes the connection with a version 5 MQCNO, the application can determine the ConnectionId from the returned MQCNO. The assigned identifier is guaranteed to be unique among all other identifiers that IBM MQ generates, such as CorrelId, MsgID, and GroupId.

Use the ConnectionId to identify long running units of work using the PCF command Inquire Connection or the MQSC command DISPLAY CONN. The ConnectionId used by MQSC commands (CONN) is derived from the ConnectionId returned here. The PCF Inquire and Stop Connection commands can use the ConnectionId returned here without modification.

You can use the ConnectionId to force the end of a long running unit of work, by specifying the ConnectionId using the PCF command Stop Connection or the MQSC command STOP CONN. See [Stop Connection](#) and [STOP CONN](#) for more information about using these commands.

This field is not returned if Version is less than MQCNO_VERSION_5.

The length of this field is given by MQ_CONNECTION_ID_LENGTH.

ConnTag (MQBYTE128)

ConnTag is a tag that the queue manager associates with the resources that are affected by the application during this connection. Each application or application instance must use a different value for the tag, so that the queue manager can correctly serialize access to the affected resources. This field is an input field, and its initial value is MQCT_NONE.

See the descriptions of the MQCNO_*_CONN_TAG_* options for further details about values to be used by different applications. The tag ceases to be valid when the application terminates or issues the MQDISC call.

Note: Connection tag values beginning with MQ in upper, lower, or mixed case in either ASCII or EBCDIC are reserved for use by IBM products. Do not use connection tag values beginning with these letters.

Use the following special value if you require no tag:

MQCT_NONE

The value is binary zero for the length of the field.

For the C programming language, the constant MQCT_NONE_ARRAY is also defined; this constant has the same value as MQCT_NONE, but is an array of characters instead of a string.

This field is used when connecting to a z/OS queue manager. In other environments, specify the value MQCT_NONE.

The length of this field is given by MQ_CONN_TAG_LENGTH. This field is ignored if *Version* is less than MQCNO_VERSION_3.

Options (MQLONG)

Options that control the action of MQCONN.

Accounting options

The following options control the type of accounting if the *AccountingConnOverride* queue manager attribute is set to MQMON_ENABLED:

MQCNO_ACCOUNTING_MQI_ENABLED

When monitoring data collection is switched off in the queue manager definition by setting the *MQIAccounting* attribute to MQMON_OFF, setting this flag enables MQI accounting data collection.

MQCNO_ACCOUNTING_MQI_DISABLED

When monitoring data collection is switched off in the queue manager definition by setting the *MQIAccounting* attribute to MQMON_OFF, setting this flag stops MQI accounting data collection.

MQCNO_ACCOUNTING_Q_ENABLED

When queue-accounting data collection is switched off in the queue manager definition by setting the *MQIAccounting* attribute to MQMON_OFF, setting this flag enables accounting data collection for those queues that specify a queue manager in the *MQIAccounting* field of their queue definition.

MQCNO_ACCOUNTING_Q_DISABLED

When queue-accounting data collection is switched off in the queue manager definition by setting the *MQIAccounting* attribute to MQMON_OFF, setting this flag switches off accounting data collection for those queues that specify a queue manager in the *MQIAccounting* field of their queue definition.

If none of these flags are defined, the accounting for the connection is as defined in the Queue Manager attributes.

Binding options

The following options control the type of IBM MQ binding to use. Specify only one of these options:

MQCNO_STANDARD_BINDING

The application and the local queue manager agent (the component that manages queuing operations) run in separate units of execution (typically, in separate processes). This arrangement maintains the integrity of the queue manager; that is, it protects the queue manager from errant programs.

If the queue manager supports multiple binding types, and you set MQCNO_STANDARD_BINDING, the queue manager uses the *DefaultBindType* attribute in the *Connection* stanza in the *qm.ini* file to select the actual type of binding. If this stanza is not defined, or the value cannot be used or is not appropriate for the application, the queue manager selects an appropriate binding type. The queue manager sets the actual binding type used in the connect options.

Use MQCNO_STANDARD_BINDING in situations where the application might not have been fully tested, or might be unreliable or untrustworthy. MQCNO_STANDARD_BINDING is the default.

This option is supported in all environments.

If you are linking to the mqm library, then a standard server connection using the default bind type is attempted first. If the underlying server library failed to load, a client connection is attempted instead.

- If the MQ_CONNECT_TYPE environment variable is specified, one of the following options can be supplied to change the behavior of MQCONN or MQCONNEX if MQCNO_STANDARD_BINDING is specified. (The exception to this is if MQCNO_FASTPATH_BINDING is specified with MQ_CONNECT_TYPE set to LOCAL or STANDARD to allow fastpath connections to be downgraded by the administrator without a related change to the application:

Value	Meaning
CLIENT	A client connection only is attempted.
FASTPATH	This value was supported in previous releases, but is now ignored if specified.
LOCAL	A server connection only is attempted. Fastpath connections are downgraded to a standard server connection.
STANDARD	Supported for compatibility with previous releases. This value is now treated as LOCAL.

- If the MQ_CONNECT_TYPE environment variable is not set when MQCONNEX is called, a standard server connection using the default bind type is attempted. If the server library fails to load, a client connection is attempted.

MQCNO_FASTPATH_BINDING

The application and the local queue manager agent are part of the same unit of execution. This is in contrast to the typical method of binding, where the application and the local queue manager agent run in separate units of execution.

MQCNO_FASTPATH_BINDING is ignored if the queue manager does not support this type of binding; processing continues as though the option had not been specified.

MQCNO_FASTPATH_BINDING can be of advantage in situations where multiple processes consume more resources than the overall resource used by the application. An application that uses the fastpath binding is known as a *trusted application*.


Consider the following important points when deciding whether to use the fastpath binding:

- Using the MQCNO_FASTPATH_BINDING option does not prevent an application altering or corrupting messages and other data areas belonging to the queue manager. Use this option only in situations where you have fully evaluated these issues.
- The application must not use asynchronous signals or timer interrupts (such as `sigkill`) with MQCNO_FASTPATH_BINDING. There are also restrictions on the use of shared memory segments.
- The application must use the MQDISC call to disconnect from the queue manager.
- The application must finish before you end the queue manager with the `endmqm` command.
- On IBM i, the job must run under a user profile that belongs to the QMQMADM group. Also, the program must not stop abnormally, otherwise unpredictable results can occur.
- On UNIX systems, the mqm user identifier must be the effective user identifier, and the mqm group identifier must be the effective group identifier. To make the application run in this way, configure the program so that it is owned by the mqm user identifier and mqm group identifier, and then set the `setuid` and `setgid` permission bits on the program.

The IBM MQ Object Authority Manager (OAM) still uses the real user ID for authority checking.

- On Windows, the user running the program must be a member of the mqm group. Fastpath binding is not supported for 32 bit applications.

The MQCNO_FASTPATH_BINDING option is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, and Windows.

 On z/OS, the option is accepted but ignored.

For more information about the implications of using trusted applications, see [Restrictions for trusted applications](#).

MQCNO_SHARED_BINDING


With MQCNO_SHARED_BINDING, the application and the local-queue-manager agent share some resources. MQCNO_SHARED_BINDING is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

MQCNO_ISOLATED_BINDING

In this case, the application process and the local queue manager agent are isolated from each other in that they do not share resources. MQCNO_ISOLATED_BINDING is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

MQCNO_CLIENT_BINDING

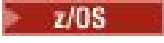
Specify this option to make the application attempt a client connection only. This option has the following limitations:

-  MQCNO_CLIENT_BINDING is ignored on z/OS.
- MQCNO_CLIENT_BINDING is rejected with MQRC_OPTIONS_ERROR if it is specified with any MQCNO binding option other than MQCNO_STANDARD_BINDING.

- MQCNO_CLIENT_BINDING is not available for Java or .NET as they have their own mechanisms for choosing the bind type.

MQCNO_LOCAL_BINDING

Specify this option to make the application attempt a server connection. If either MQCNO_FASTPATH_BINDING, MQCNO_ISOLATED_BINDING, or MQCNO_SHARED_BINDING is also specified, then the connection is of that type instead, and is documented in this section. Otherwise a standard server connection is attempted using the default bind type. MQCNO_LOCAL_BINDING has the following limitations:

-  MQCNO_LOCAL_BINDING is ignored on z/OS.
- MQCNO_LOCAL_BINDING is rejected with MQRC_OPTIONS_ERROR if it is specified with any MQCNO reconnect option other than MQCNO_RECONNECT_AS_DEF.
- MQCNO_LOCAL_BINDING is not available for Java or .NET as they have their own mechanisms for choosing the bind type.

On AIX, HP-UX, Solaris, Linux, and Windows, you can use the environment variable MQ_CONNECT_TYPE with the bind type specified by the *Options* field, to control the type of binding used. If you specify this environment variable, it must have the value FASTPATH or STANDARD ; if it has a different value, it is ignored. The value of the environment variable is case sensitive; see [MQCONN environment variable](#) for more information.

The environment variable and *Options* field interact as follows:

- If you omit the environment variable, or give it a value that is not supported, use of the fastpath binding is determined solely by the *Options* field.
- If you give the environment variable a supported value, the fastpath binding is used only if *both* the environment variable and *Options* field specify the fastpath binding.

Connection-tag options



These options are supported only when connecting to a z/OS queue manager and they control the use of the connection tag *ConnTag* . You can specify only one of these options:

MQCNO_SERIALIZE_CONN_TAG_Q_MGR

This option requests exclusive use of the connection tag within the local queue manager. If the connection tag is already in use in the local queue manager, the MQCONN call fails with reason code MQRC_CONN_TAG_IN_USE. The outcome of the call is not affected by using the connection tag elsewhere in the queue-sharing group to which the local queue manager belongs.

MQCNO_SERIALIZE_CONN_TAG_QSG

This option requests exclusive use of the connection tag within the queue-sharing group to which the local queue manager belongs. If the connection tag is already in use in the queue-sharing group, the MQCONN call fails with reason code MQRC_CONN_TAG_IN_USE.

MQCNO_RESTRICT_CONN_TAG_Q_MGR

This option requests shared use of the connection tag within the local queue manager. If the connection tag is already in use in the local queue manager, the MQCONN call can succeed if the requesting application is running in the same processing scope as the existing user of the tag. If this condition is not satisfied, the MQCONN call fails with reason code MQRC_CONN_TAG_IN_USE. The outcome of the call is not affected by use of the connection tag elsewhere in the queue-sharing group to which the local queue manager belongs.

- Applications must run within the same MVS address space to share the connection tag. If the application using the connection tag is a client application, MQCNO_RESTRICT_CONN_TAG_Q_MGR is not allowed.

MQCNO_RESTRICT_CONN_TAG_QSG

This option requests shared use of the connection tag within the queue-sharing group to which the local queue manager belongs. If the connection tag is already in use in the queue-sharing group, the MQCONN call can succeed provided the requesting application is running in the same processing scope and is connected to the same queue manager, as the existing user of the tag.

If these conditions are not satisfied, the MQCONN call fails with reason code MQRC_CONN_TAG_IN_USE.

- Applications must run within the same MVS address space to share the connection tag. If the application using the connection tag is a client application, MQCNO_RESTRICT_CONN_TAG_QSG is not allowed.

If none of these options are specified, *ConnTag* is not used. These options are not valid if *Version* is less than MQCNO_VERSION_3.

Handle-sharing options

These options are supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, and Windows. They control the sharing of handles between different threads (units of parallel processing) within the same process. You can specify only one of these options:

MQCNO_HANDLE_SHARE_NONE

This option indicates that connection and object handles can be used only by the thread that caused the handle to be allocated (that is, the thread that issued the MQCONN, MQCONN, or MQOPEN call). The handles cannot be used by other threads belonging to the same process.

MQCNO_HANDLE_SHARE_BLOCK

This option indicates that connection and object handles allocated by one thread of a process can be used by other threads belonging to the same process. However, only one thread at a time can use any particular handle; that is, only serial use of a handle is permitted. If a thread tries to use a handle that is already in use by another thread, the call blocks (waits) until the handle becomes available.

MQCNO_HANDLE_SHARE_NO_BLOCK

This is the same as MQCNO_HANDLE_SHARE_BLOCK, except that if the handle is in use by another thread, the call completes immediately with MQCC_FAILED and MQRC_CALL_IN_PROGRESS instead of blocking until the handle becomes available.

A thread can have zero or one non-shared handles:

- Each MQCONN or MQCONN call that specifies MQCNO_HANDLE_SHARE_NONE returns a new nonshared handle on the first call, and the same non-shared handle on the second and later calls (assuming no intervening MQDISC call). The reason code is MQRC_ALREADY_CONNECTED for the second and later calls.
- Each MQCONN call that specifies MQCNO_HANDLE_SHARE_BLOCK or MQCNO_HANDLE_SHARE_NO_BLOCK returns a new shared handle on each call.

Object handles inherit the same sharing properties as the connection handle specified on the MQOPEN call that created the object handle. Also, units of work inherit the same sharing properties as the connection handle used to start the unit of work; if the unit of work is started in one thread using a shared handle, the unit of work can be updated in another thread using the same handle.

If you do not specify a handle-sharing option, the default is determined by the environment:

- In the Microsoft Transaction Server (MTS) environment, the default is the same as MQCNO_HANDLE_SHARE_BLOCK.
- In other environments, the default is the same as MQCNO_HANDLE_SHARE_NONE.

Reconnection options

Reconnection options determine if a connection is reconnectable. Only client connections are reconnectable.

MQCNO_RECONNECT_AS_DEF

The reconnection option is resolved to its default value. If no default is set, the value of this option resolves to DISABLED. The value of the option is passed to the server, and can be queried by PCF and MQSC.

MQCNO_RECONNECT

The application can be reconnected to any queue manager consistent with the value of the *QmgrName* parameter of MQCONN. Use the MQCNO_RECONNECT option only if there is no affinity between the client application and the queue manager with which it initially established a connection. The value of the option is passed to the server, and can be queried by PCF and MQSC.

MQCNO_RECONNECT_DISABLED

The application cannot be reconnected. The value of the option is not passed to the server.

MQCNO_RECONNECT_Q_MGR

The application can be reconnected only to the queue manager with which it originally connected. Use this value if a client can be reconnected, but there is an affinity between the client application and the queue manager with which it originally established a connection. Choose this value if you want a client to automatically reconnect to the standby instance of a highly available queue manager. The value of the option is passed to the server, and can be queried by PCF and MQSC.

Use the options MQCNO_RECONNECT, MQCNO_RECONNECT_DISABLED and MQCNO_RECONNECT_Q_MGR only for client connections. If the options are used for a binding connection, MQCONN fails with completion code MQCC_FAILED and reason code MQRC_OPTIONS_ERROR. Automatic client reconnect is not supported by IBM MQ classes for Java

Conversation-sharing options

The following options apply only to TCP/IP client connections. For SNA, SPX and NetBios channels, these values are ignored and the channel runs as in previous versions of the product

MQCNO_NO_CONV_SHARING

This option does not permit conversation sharing.

You might use MQCNO_NO_CONV_SHARING in situations where conversations are heavily loaded and, therefore, where contention is a possibility on the server-connection end of the channel instance on which the sharing conversations exist. MQCNO_NO_CONV_SHARING behaves like *sharecnv(1)* when connected to a channel that supports conversation sharing, and *sharecnv(0)* when connected to a channel that does not support conversation sharing.

MQCNO_ALL_CONVS_SHARE

This option permits conversation sharing; the application does not place any limit on the number of connections on the channel instance. This option is the default value.

If the application indicates that the channel instance can share, but the *SharingConversations* (SHARECNV) definition on the server-connection end of the channel is set to one, no sharing occurs and no warning is given to the application.

Similarly, if the application indicates that sharing is permitted but the server-connection *SharingConversations* definition is set to zero, no warning is given, and the application exhibits the same behavior as a client in versions of the product earlier than Version 7.0; the application setting relating to sharing conversations is ignored.

MQCNO_NO_CONV_SHARING and MQCNO_ALL_CONVS_SHARE are mutually exclusive. If both options are specified on a particular connection, the connection is rejected with a reason code of MQRC_OPTIONS_ERROR.

Channel definition options

The following options control the use of the channel definition structure passed in the MQCNO:

MQCNO_CD_FOR_OUTPUT_ONLY

This option permits channel definition structure in the MQCNO to be used only to return the channel name used on a successful MQCONN call.

If a valid channel definition structure is not provided, the call fails with the reason code MQRC_CD_ERROR.

If the application is not running as a client, the option is ignored.

The returned channel name can be used on a subsequent MQCONN call using the MQCNO_USE_CD_SELECTION option to reconnect using the same channel definition. This can be useful when there are multiple applicable channel definitions in the client channel table.

MQCNO_USE_CD_SELECTION

This option permits MQCONN call to connect using the channel name contained in the channel definition structure passed in the MQCNO.

If the MQSERVER environment variable is set, the channel definition defined by it is used. If MQSERVER is not set, the client channel table is used.

If a channel definition with matching channel name and queue manager name is not found, the call fails with reason code MQRC_Q_MGR_NAME_ERROR.

If a valid channel definition structure is not provided, the call fails with the reason code MQRC_CD_ERROR.

If the application is not running as a client, the option is ignored.

Default option

If you require none of the options described above, you can use the following option:

MQCNO_NONE

No options are specified.

Use MQCNO_NONE to aid program documentation. It is not intended that this option is used with any other MQCNO_* option, but because its value is zero, such use cannot be detected.

SecurityParmsOffset (MQLONG)

SecurityParmsOffset is the offset in bytes of the MQCSP structure from the start of the MQCNO structure. The offset can be positive or negative. This field is an input field, with an initial value of 0.

This field is ignored if *Version* is less than MQCNO_VERSION_5.

The MQCSP structure is defined in [“MQCSP - Security parameters” on page 320](#).

SecurityParmsPtr (PMQCSP)

SecurityParmsPtr is the address of the MQCSP structure, used to specify a user ID and password for authentication by the authorization service. This field is an input field, and its initial value is a null pointer or null bytes.

This field is ignored if *Version* is less than MQCNO_VERSION_5.

The MQCSP structure is defined in [“MQCSP - Security parameters” on page 320](#).

SSLConfigOffset (MQLONG)

SSLConfigOffset is the offset in bytes of an MQSCO structure from the start of the MQCNO structure. The offset can be positive or negative. This field is an input field, with an initial value of 0.

Use *SSLConfigOffset* only when the application issuing the MQCONN call is running as an IBM MQ MQI client. For information about how to use this field, see the description of the *SSLConfigPtr* field.

This field is ignored if *Version* is less than MQCNO_VERSION_4.

SSLConfigPtr (PMQSCO)

SSLConfigPtr is an input field. Its initial value is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Use *SSLConfigPtr* and *SSLConfigOffset* only when the application issuing the MQCONN call is running as an IBM MQ MQI client and the channel protocol is TCP/IP. If the application is not running as an IBM MQ client, or the channel protocol is not TCP/IP, *SSLConfigPtr* and *SSLConfigOffset* are ignored.

By specifying *SSLConfigPtr* or *SSLConfigOffset*, plus either *ClientConnPtr* or *ClientConnOffset*, the application can control the use of SSL for the client connection. When the SSL information is specified in this way, the environment variables MQSSLKEYR and MQSSLCRY are ignored; any SSL-related information in the client channel definition table (CCDT) is also ignored.

The SSL information can be specified only on:

- The first MQCONN call of the client process, or
- A subsequent MQCONN call when all previous SSL/TLS connections to the queue manager have been concluded using MQDISC.

These are the only states in which the process-wide SSL environment can be initialized. If an MQCONN call is issued specifying SSL information when the SSL environment already exists, the SSL information on the call is ignored and the connection is made using the existing SSL environment; the call returns completion code MQCC_WARNING and reason code MQRC_SSL_ALREADY_INITIALIZED in this case.

You can provide the MQSCO structure in the same way as the MQCD structure, either by specifying an address in *SSLConfigPtr*, or by specifying an offset in *SSLConfigOffset*; see the description of *ClientConnPtr* for details of how to do this. However, you can use no more than one of *SSLConfigPtr* and *SSLConfigOffset*; the call fails with reason code MQRC_SSL_CONFIG_ERROR if both are nonzero.

Once the MQCONN call has completed, the MQSCO structure is not referenced again.

This field is ignored if *Version* is less than MQCNO_VERSION_4.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

StrucId (MQCHAR4)

StrucId is always an input field. Its initial value is MQCNO_STRUC_ID.

The value must be:

MQCNO_STRUC_ID

Identifier for connect-options structure.

For the C programming language, the constant MQCNO_STRUC_ID_ARRAY is also defined; this constant has the same value as MQCNO_STRUC_ID, but is an array of characters instead of a string.

Version (MQLONG)

Version is always an input field. Its initial value is MQCNO_VERSION_1.

The value must be one of the following:

MQCNO_VERSION_1

Version-1 connect-options structure.

MQCNO_VERSION_2

Version-2 connect-options structure.

MQCNO_VERSION_3

Version-3 connect-options structure.

MQCNO_VERSION_4

Version-4 connect-options structure.

MQCNO_VERSION_5

Version-5 connect-options structure.

This version of the MQCNO structure extends MQCNO_VERSION_3 on z/OS, and MQCNO_VERSION_4 on all other platforms.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQCNO_CURRENT_VERSION

Current version of connect-options structure.

Initial values and language declarations for MQCNO

Table 477. Initial values of fields in MQCNO for MQCNO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCNO_STRUC_ID	'CNO↯'
<i>Version</i>	MQCNO_VERSION_1	1
<i>Options</i>	MQCNO_NONE	0
<i>ClientConnOffset</i>	None	0
<i>ClientConnPtr</i>	None	Null pointer or null bytes
<i>ConnTag</i>	MQCT_NONE	Nulls
<i>SSLConfigPtr</i>	None	Null pointer or null bytes
<i>SSLConfigOffset</i>	None	0
<i>ConnectionId</i>	None	Null pointer or null bytes
<i>SecurityParmsOffset</i>	None	Null pointer or null bytes
<i>SecurityParmsPtr</i>	None	Null pointer or null bytes
Notes:		
1. The symbol ↯ represents a single blank character.		
2. In the C programming language, the macro variable MQCNO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:		
<pre>MQCNO MyCNO = {MQCNO_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQCNO MQCNO;
struct tagMQCNO {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     Options;          /* Options that control the action of
                                MQCONN */
    MQLONG     ClientConnOffset; /* Offset of MQCD structure for client
                                connection */
}
```

```

MQPTR      ClientConnPtr;      /* Address of MQCD structure for client
                                connection */
MQBYTE128  ConnTag;            /* Queue-manager connection tag */
PMQSCO     SSLConfigPtr;       /* Address of MQSCO structure for client
                                connection */
MQLONG     SSLConfigOffset;     /* Offset of MQSCO structure for client
                                connection */
MQBYTE24   ConnectionId;       /* Unique connection identifier */
MQLONG     SecurityParmsOffset /* Security fields */
PMQCSP     SecurityParmsPtr    /* Security parameters */
};

```

COBOL declaration

```

**  MQCNO structure
10 MQCNO.
**    Structure identifier
15 MQCNO-STRUCID      PIC X(4).
**    Structure version number
15 MQCNO-VERSION     PIC S9(9) BINARY.
**    Options that control the action of MQCONN
15 MQCNO-OPTIONS     PIC S9(9) BINARY.
**    Offset of MQCD structure for client connection
15 MQCNO-CLIENTCONNOFFSET PIC S9(9) BINARY.
**    Address of MQCD structure for client connection
15 MQCNO-CLIENTCONNPTR  POINTER.
**    Queue-manager connection tag
15 MQCNO-CONNTAG     PIC X(128).
**    Address of MQSCO structure for client connection
15 MQCNO-SSLCONFIGPTR  POINTER.
**    Offset of MQSCO structure for client connection
15 MQCNO-SSLCONFIGOFFSET PIC S9(9) BINARY.
**    Unique connection identifier
15 MQCNO-CONNECTIONID  PIC X(24).
**    Offset of MQCSP structure for security parameters
15 MQCNO-SECURITYPARMSOFFSET PIC S9(9) BINARY.
**    Address of MQCSP structure for security parameters
15 MQCNO-SECURITYPARMSPTR POINTER.

```

PL/I declaration

```

dcl
1 MQCNO based,
3 StrucId      char(4),          /* Structure identifier */
3 Version      fixed bin(31),    /* Structure version number */
3 Options      fixed bin(31),    /* Options that control the action
                                of MQCONN */
3 ClientConnOffset fixed bin(31), /* Offset of MQCD structure for
                                client connection */
3 ClientConnPtr  pointer,        /* Address of MQCD structure for
                                client connection */
3 ConnTag       char(128),       /* Queue-manager connection tag */
3 SSLConfigPtr  pointer,        /* Address of MQSCO structure for
                                client connection */
3 SSLConfigOffset fixed bin(31), /* Offset of MQSCO structure for
                                client connection */
3 ConnectionId  char(24),        /* Unique connection identifier */
3 SecurityParmsOffset fixed bin(31); /* Offset of MQCSP structure for
                                security parameters */
3 SecurityParmsPtr pointer,      /* Address of MQCSP structure for
                                security parameters */

```

High Level Assembler declaration

MQCNO	DSECT		
MQCNO_STRUCID	DS	CL4	Structure identifier
MQCNO_VERSION	DS	F	Structure version number
MQCNO_OPTIONS	DS	F	Options that control the action of MQCONN
* MQCNO_CLIENTCONNOFFSET	DS	F	Offset of MQCD structure for client connection
* MQCNO_CLIENTCONNPTR	DS	F	Address of MQCD structure for client connection
* MQCNO_CONNTAG	DS	XL128	Queue-manager connection tag
* MQCNO_CONNECTIONID	DS	XL24	Unique connection identifier

```

*
MQCNO_SSLCONFIGOFFSET DS F      Offset of MQCSP structure for security
*                        parameters
MQCNO_SSLCONFIGPTR     DS F      Address of MQCSP structure for security
*                        parameters
MQCNO_LENGTH           EQU *-MQCNO
ORG MQCNO
MQCNO_AREA             DS CL(MQCNO_LENGTH)

```

Visual Basic declaration

```

Type MQCNO
  StrucId           As String*4 'Structure identifier'
  Version           As Long      'Structure version number'
  Options           As Long      'Options that control the action of'
                                'MQCONN'
  ClientConnOffset As Long      'Offset of MQCD structure for client'
                                'connection'
  ClientConnPtr     As MQPTR     'Address of MQCD structure for client'
                                'connection'
  ConnTag           As MQBYTE128 'Queue-manager connection tag'
  SSLConfigPtr      As MQPTR     'Address of MQSCO structure for client'
                                'connection'
  SSLConfigOffset   As Long      'Offset of MQSCO structure for client'
                                'connection'
  ConnectionId      As MQBYTE24 'Unique connection identifier'
  SecurityParmsOffset As Long    'Offset of MQCSP structure for security'
                                'parameters'
  SecurityParmsPtr  As MQPTR     'Address of MQCSP structure for security'
                                'parameters'
End Type

```

MQCSP - Security parameters

The following table summarizes the fields in the structure.

Warning: In some cases, the password in an MQCSP structure for a client application will be sent across a network in plain text. To ensure that client application passwords are protected appropriately, see [IBM MQCSP password protection](#).

Table 478. Fields in MQCSP		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>AuthenticationType</i>	Type of authentication	AuthenticationType
<i>Reserved1</i>	Required for pointer alignment on IBM i	Reserved1
<i>CSPUserIdPtr</i>	Address of user ID	CSPUserIdPtr
<i>CSPUserIdOffset</i>	Offset of user ID	CSPUserIdOffset
<i>CSPUserIdLength</i>	Length of user ID	CSPUserIdLength
<i>Reserved2</i>	Required for pointer alignment on IBM i	Reserved2
<i>CSPPasswordPtr</i>	Address of password	CSPPasswordPtr
<i>CSPPasswordOffset</i>	Offset of password	CSPPasswordOffset
<i>CSPPasswordLength</i>	Length of password	CSPPasswordLength

Overview for MQCSP

Availability: All IBM MQ products.

Purpose: The MQCSP structure enables the authorization service to authenticate a user ID and password. You specify the MQCSP connection security parameters structure on an MQCONN call.

Warning: In some cases, the password in an MQCSP structure for a client application will be sent across a network in plain text. To ensure that client application passwords are protected appropriately, see [IBM MQCSP password protection](#).

Character set and encoding: Data in MQCSP must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively.

Fields for MQCSP

The MQCSP structure contains the following fields; the fields are described in **alphabetical order**:

AuthenticationType (MQLONG)

AuthenticationType is an input field. Its initial value is MQCSP_AUTH_NONE.

This is the type of authentication to perform. Valid values are:

MQCSP_AUTH_NONE

Do not use user ID and password fields.

MQCSP_AUTH_USER_ID_AND_PWD

Authenticate user ID and password fields.

The default value is MQCSP_AUTH_NONE. With the default setting, no password protection is done.

If you require authentication, you have to set **MQCSP.AuthenticationType** to MQCSP_AUTH_USER_ID_AND_PWD.

See [MQCSP password protection](#) for more information.

CSPPasswordLength (MQLONG)

This field is the length of the password to be used in authentication.

The maximum length of the password is MQ_CSP_PASSWORD_LENGTH. If the length of the password is greater than the maximum length permitted, the authentication request fails with MQRC_NOT_AUTHORIZED.

The value of MQ_CSP_PASSWORD_LENGTH is 256.

This field is an input field. The initial value of this field is 0.

CSPPasswordOffset (MQLONG)

This is the offset in bytes of the password to be used in authentication. The offset can be positive or negative.

This is an input field. The initial value of this field is 0.

CSPPasswordPtr (MQPTR)

This is the address in bytes of the password to be used in authentication.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQCNO_VERSION_5.

This field can contain an empty password which is rejected by the operating system or LDAP password checking, depending on setup, but is not rejected by IBM MQ before passing it the authentication method.

CSPUserIdLength (MQLONG)

This field is the length of the user ID to be used in authentication.

The maximum length of the user ID is dependent on the platform, see [User IDs](#). If the length of the user ID is greater than the maximum length permitted, the authentication request fails with MQRC_NOT_AUTHORIZED.

This field is an input field. The initial value of this field is 0.

CSPUserIdOffset (MQLONG)

This is the offset in bytes of the user ID to be used in authentication. The offset can be positive or negative.

This is an input field. The initial value of this field is 0.

CSPUserIdPtr (MQPTR)

This is the address in bytes of the user ID to be used in authentication.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQCNO_VERSION_5.

This field can contain an operating system user ID when an **AUTHTYPE** of *IDPWOS* is named in the CONNAUTH field of the queue manager.

On Windows this can be a fully qualified domain user ID.

This field can contain an LDAP User ID when an **AUTHTYPE** of *IDPWLDAP* is named in the CONNAUTH field of the queue manager.

Reserved1 (MQBYTE4)

A reserved field, required for pointer alignment on IBM i.

This is an input field. The initial value of this field is all null.

Reserved2 (MQBYTE8)

A reserved field, required for pointer alignment on IBM i.

This is an input field. The initial value of this field is all null.

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQCSP_STRUC_ID

Identifier for the security parameters structure.

For the C programming language, the constant MQCSP_STRUC_ID_ARRAY is also defined; this has the same value as MQCSP_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQCSPSTRUC_ID.

Version (MQLONG)

Structure version number.

The value must be:

MQCSP_VERSION_1

Version-1 security parameters structure.

The following constant specifies the version number of the current version:

MQCSP_CURRENT_VERSION

Current version of security parameters structure.

This is always an input field. The initial value of this field is MQCSP_VERSION_1.

Initial values and language declarations for MQCSP

Table 479. Initial values of fields in MQCSP for MQCSP

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCSP_STRUC_ID	'CSP~'
<i>Version</i>	MQCSP_VERSION_1	1
<i>AuthenticationType</i>	None	MQCSP_AUTH_NONE
<i>Reserved1</i>	None	Null string or blanks
<i>CSPUserIdPtr</i>	None	Null pointer or null bytes
<i>CSPUserIdOffset</i>	None	0
<i>CSPUserIdLength</i>	None	0
<i>Reserved2</i>	None	Null string or blanks
<i>CSPPasswordPtr</i>	None	Null pointer or null bytes
<i>CSPPasswordOffset</i>	None	0
<i>CSPPasswordLength</i>	None	0

Notes:

1. The symbol ~ represents a single blank character.
2. In the C programming language, the macro variable MQCSP_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure:

```
MQCSP MyCSP = {MQCSP_DEFAULT};
```

C declaration

```
typedef struct tagMQCSP MQCSP;
struct tagMQCSP {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     AuthenticationType; /* Type of authentication */
    MQBYTE4    Reserved1;        /* Required for IBM i pointer
                                alignment */
    MQPTR      CSPUserIdPtr;      /* Address of user ID */
    MQLONG     CSPUserIdOffset;   /* Offset of user ID */
    MQLONG     CSPUserIdLength;   /* Length of user ID */
    MQBYTE8    Reserved2;        /* Required for IBM i pointer
                                alignment */
    MQPTR      CSPPasswordPtr;    /* Address of password */
    MQLONG     CSPPasswordOffset; /* Offset of password */
    MQLONG     CSPPasswordLength; /* Length of password */
};
```

COBOL declaration

```
** MQCSP structure
10 MQCSP.
** Structure identifier
15 MQCSP-STRUCID PIC X(4).
** Structure version number
15 MQCSP-VERSION PIC S9(9) BINARY.
** Type of authentication
15 MQCSP-AUTHENTICATIONTYPE PIC S9(9) BINARY.
** Required for IBM i pointer alignment
15 MQCSP-RESERVED1 PIC X(4).
** Address of user ID
15 MQCSP-CSPUSERIDPTR POINTER.
```

```

**      Offset of user ID
15 MQCSP-CSPUSERIDOFFSET PIC S9(9) BINARY.
**      Length of user ID
15 MQCSP-CSPUSERIDLENGTH PIC S9(9) BINARY.
**      Required for IBM i pointer alignment
15 MQCSP-RESERVED2       PIC X(4).
**      Address of password
15 MQCSP-CSPPASSWORDPTR  POINTER.
**      Offset of password
15 MQCSP-CSPPASSWORDOFFSET PIC S9(9) BINARY.
**      Length of password
15 MQCSP-CSPPASSWORDLENGTH PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQCSP based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31), /* Structure version number */
    3 AuthenticationType fixed bin(31), /* Type of authentication */
    3 Reserved1        char(4),          /* Required for IBM i pointer
                                         alignment */
    3 CSPUserIdPtr     pointer,          /* Address of user ID */
    3 CSPUserIdOffset  fixed bin(31), /* Offset of user ID */
    3 CSPUserIdLength  fixed bin(31), /* Length of user ID */
    3 Reserved2        char(8),          /* Required for IBM i pointer
                                         alignment */
    3 CSPPasswordPtr   pointer,          /* Address of password */
    3 CSPPasswordOffset fixed bin(31), /* Offset of user ID */
    3 CSPPasswordLength fixed bin(31); /* Length of user ID */

```

Visual Basic declaration

```

Type MQCSP
  StrucId          As String*4  'Structure identifier'
  Version          As Long      'Structure version number'
  AuthenticationType As Long     'Type of authentication'
  Reserved1        As MBYTE4    'Required for IBM i pointer'
                                     'alignment'
  CSPUserIdPtr     As MQPTR      'Address of user ID'
  CSPUserIdOffset  As Long      'Offset of user ID'
  CSPUserIdLength  As Long      'Length of user ID'
  Reserved2        As MBYTE8    'Required for IBM i pointer'
                                     'alignment'
  CSPPasswordPtr   As MQPTR      'Address of password'
  CSPPasswordOffset As Long      'Offset of password'
  CSPPasswordLength As Long      'Length of password'
End Type

```

MQCTLO - Control callback options structure

The following table summarizes the fields in the structure. Structure specifying the control callback function.

Table 480. Fields in MQCTLO		
Field	Description	Topic
<i>StrucID</i>	Structure identifier	StrucID
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options
<i>Reserved</i>	Reserved field	Options
<i>ConnectionArea</i>	Field for callback function to use	ConnectionArea

Overview for MQCTLO

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS, and IBM MQ MQI clients connected to these systems. Overview of the MQCTLO structure.

Purpose: The MQCTLO structure is used to specify options relating to a control callbacks function.

The structure is an input and output parameter on the [MQCTL](#) call.

Version: The current version of MQCTLO is MQCTLO_VERSION_1.

Character set and encoding: Data in MQCTLO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQCTLO

Alphabetic list of fields for the MQCTLO structure.

The MQCTLO structure contains the following fields; the fields are described in alphabetical order:

ConnectionArea (MQPTR)

Control options structure - ConnectionArea field

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged to the [ConnectionArea](#) field in the MQCBC structure, which is an input parameter to the callback.

This field is ignored for all operations other than MQOP_START and MQOP_START_WAIT.

This is an input and output field to the callback function. The initial value of this field is a null pointer or null bytes.

Options (MQLONG)

Control options structure - Options field

Options that control the action of MQCTL.

MQCTLO_FAIL_IF QUIESCING

Force the MQCTL call to fail if the queue manager or connection is in the quiescing state.

Specify MQGMO_FAIL_IF QUIESCING, in the MQGMO options passed on the MQCB call, to cause notification to message consumers when they are quiescing.

MQCTLO_THREAD_AFFINITY

This option informs the system that the application requires that all message consumers, for the same connection, are called on the same thread. This thread will be used for all invocations of the consumers until the connection is stopped.

Default option: If you do not need any of the options described, use the following option:

MQCTLO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values. MQCTLO_NONE is defined to aid program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *Options* field is MQCTLO_NONE.

Reserved (MQLONG)

This is a reserved field. The value must be zero.

StrucId (MQCHAR4)

Control options structure - StrucId field

This is the structure identifier; the value must be:

MQCTLO_STRUC_ID

Identifier for Control Options structure.

For the C programming language, the constant MQCTLO_STRUC_ID_ARRAY is also defined; this has the same value as MQCTLO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQCTLO_STRUC_ID.

Version (MQLONG)

Control options structure - Version field

This is the structure version number; the value must be:

MQCTLO_VERSION_1

Version-1 Control options structure.

The following constant specifies the version number of the current version:

MQCTLO_CURRENT_VERSION

Current version of Control options structure.

This is always an input field. The initial value of this field is MQCTLO_VERSION_1.

Initial values and language declarations for MQCTLO

Control options structure - Initial values

Table 481. Initial values of fields in MQCTLO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQCTLO_STRUC_ID	'CTLO'
<i>Version</i>	MQCTLO_VERSION_1	1
<i>Options</i>	MQCTLO_NONE	Nulls
<i>Reserved</i>	Reserved field	
<i>ConnectionArea</i>	None	Null pointer or null bytes
Notes:		
1. In the C programming language, the macro variable MQCTLO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:		
<pre>MQCTLO MyCTLO = {MQCTLO_DEFAULT};</pre>		

C declaration

Control Options structure - C language declaration

```
typedef struct tagMQCTLO MQCTLO;
struct tagMQCTLO {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;          /* Structure version number */
    MQLONG    Options;          /* Options that control the action of MQCTL */
    MQLONG    Reserved;         /* Reserved field */

    MQPTR     ConnectionArea; /* Connection work area passed to the function */
};
```

COBOL declaration

```
** MQCTLO structure
10  MQCTLO.
** Structure Identifier
15  MQCTLO-STRUCID                PIC X(4).
** Structure Version
```

```

15 MQCTLO-VERSION          PIC S9(9) BINARY.
** Options
15 MQCTLO-OPTIONS         PIC S9(9) BINARY.
** Reserved
15 MQCTLO-RESERVED        PIC S9(9) BINARY.
** ConnectionArea
15 MQCTLO-CONNECTIONAREA  POINTER

```

PL/I declaration

```

dcl
  1 MQCTLO based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version */
    3 Options          fixed bin(31),    /* Options */
    3 Reserved         fixed bin(31),
    3 ConnectionArea   pointer;          /* Connection work area */

```

MQDH - Distribution header

The following table summarizes the fields in the structure.

Table 482. Fields in MQDH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQDH structure plus following records	StrucLength
<i>Encoding</i>	Numeric encoding of data that follows array of MQPMR records	Encoding
<i>CodedCharSetId</i>	Character set identifier of data that follows array of MQPMR records	CodedCharSetId
<i>Format</i>	Format name of data that follows array of MQPMR records	Format
<i>Flags</i>	General flags	Flags
<i>PutMsgRecFields</i>	Flags indicating which MQPMR fields are present	PutMsgRecFields
<i>RecsPresent</i>	Number of object records present	RecsPresent
<i>ObjectRecOffset</i>	Offset of first object record from start of MQDH	ObjectRecOffset
<i>PutMsgRecOffset</i>	Offset of first put-message record from start of MQDH	PutMsgRecOffset

Overview for MQDH

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

Purpose: The MQDH structure describes the additional data that is present in a message when that message is a distribution-list message stored on a transmission queue. A distribution-list message is a message that is sent to multiple destination queues. The additional data consists of the MQDH structure followed by an array of MQOR records and an array of MQPMR records.

This structure is used by specialized applications that put messages directly on transmission queues, or that remove messages from transmission queues (for example: message channel agents).

Applications that want to put messages to distribution lists must not use this structure. Instead, they must use the MQOD structure to define the destinations in the distribution list, and the MQPMO structure to specify message properties or receive information about the messages sent to the individual destinations.

Format name: MQFMT_DIST_HEADER.

Character set and encoding: Data in MQDH must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE.

Set the character set and encoding of the MQDH into the *CodedCharSetId* and *Encoding* fields in:

- The MQMD (if the MQDH structure is at the start of the message data), or
- The header structure that precedes the MQDH structure (all other cases).

Usage: When an application puts a message to a distribution list, and some or all of the destinations are remote, the queue manager prefixes the application message data with the MQXQH and MQDH structures, and places the message on the relevant transmission queue. The data therefore occurs in the following sequence when the message is on a transmission queue:

- MQXQH structure
- MQDH structure plus arrays of MQOR and MQPMR records
- Application message data

Depending on the destinations, the queue manager can generate more than one such message, and place it on different transmission queues. In this case, the MQDH structures in those messages identify different subsets of the destinations defined by the distribution list opened by the application.

An application that puts a distribution-list message directly on a transmission queue must conform to the sequence described earlier, and must ensure that the MQDH structure is correct. If the MQDH structure is not valid, the queue manager can fail the MQPUT or MQPUT1 call with reason code MQRC_DH_ERROR.

You can store messages on a queue in distribution-list form only if you have defined the queue as being able to support distribution list messages (see the *DistLists* queue attribute described in [“Attributes for queues” on page 801](#)). If an application puts a distribution-list message directly on a queue that does not support distribution lists, the queue manager splits the distribution list message into individual messages, and places those on the queue instead.

Fields for MQDH

The MQDH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This is the character set identifier of the data that follows the arrays of MQOR and MQPMR records; it does not apply to character data in the MQDH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. You can use the following special value:

MQCCSI_INHERIT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the MQGET call does not return the value MQCCSI_INHERIT.

You cannot use MQCCSI_INHERIT if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

This value is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

The initial value of this field is MQCCSI_UNDEFINED.

Encoding (MQLONG)

This is the numeric encoding of the data that follows the arrays of MQOR and MQPMR records; it does not apply to numeric data in the MQDH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

Flags (MQLONG)

You can specify the following flag:

MQDHF_NEW_MSG_IDS

Generate a new message identifier for each destination in the distribution list. Set this only when there are no put-message records present, or when the records are present but they do not contain the *MsgId* field.

Using this flag defers generation of the message identifiers until the moment when the distribution-list message is finally split into individual messages. This minimizes the amount of control information that must flow with the distribution-list message.

When an application puts a message to a distribution list, the queue manager sets MQDHF_NEW_MSG_IDS in the MQDH that it generates when both of the following are true:

- There are no put-message records provided by the application, or the records provided do not contain the *MsgId* field.
- The *MsgId* field in MQMD is MQMI_NONE, or the *Options* field in MQPMO includes MQPMO_NEW_MSG_ID

If no flags are needed, specify the following:

MQDHF_NONE

No flags have been specified. MQDHF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQDHF_NONE.

Format (MQCHAR8)

This is the format name of the data that follows the arrays of MQOD and MQPMR records (whichever occurs last).

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *Format* field in MQMD.

The initial value of this field is MQFMT_NONE.

ObjectRecOffset (MQLONG)

This gives the offset in bytes of the first record in the array of MQOR object records containing the names of the destination queues. There are *RecsPresent* records in this array. These records (plus any bytes skipped between the first object record and the previous field) are included in the length given by the *StrucLength* field.

A distribution list must always contain at least one destination, so *ObjectRecOffset* must always be greater than zero.

The initial value of this field is 0.

PutMsgRecFields (MQLONG)

You can specify none or more of the following flags:

MQPMRF_MSG_ID

Message-identifier field is present.

MQPMRF_CORREL_ID

Correlation-identifier field is present.

MQPMRF_GROUP_ID

Group-identifier field is present.

MQPMRF_FEEDBACK

Feedback field is present.

MQPMRF_ACCOUNTING_TOKEN

Accounting-token field is present.

If no MQPMR fields are present, specify the following:

MQPMRF_NONE

No put-message record fields are present. MQPMRF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQPMRF_NONE.

PutMsgRecOffset (MQLONG)

This gives the offset in bytes of the first record in the array of MQPMR put message records containing the message properties. If present, there are *RecsPresent* records in this array. These records (plus any bytes skipped between the first put message record and the previous field) are included in the length given by the *StrucLength* field.

Put message records are optional; if no records are provided, *PutMsgRecOffset* is zero, and *PutMsgRecFields* has the value MQPMRF_NONE.

The initial value of this field is 0.

RecsPresent (MQLONG)

This is the number of destinations. A distribution list must always contain at least one destination, so *RecsPresent* must always be greater than zero.

The initial value of this field is 0.

StrucId (MQCHAR4)

The value must be:

MQDH_STRUC_ID

Identifier for distribution header structure.

For the C programming language, the constant MQDH_STRUC_ID_ARRAY is also defined; this has the same value as MQDH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQDH_STRUC_ID.

StrucLength (MQLONG)

This is the number of bytes from the start of the MQDH structure to the start of the message data following the arrays of MQOR and MQPMR records. The data occurs in the following sequence:

- MQDH structure
- Array of MQOR records
- Array of MQPMR records
- Message data

The arrays of MQOR and MQPMR records are addressed by offsets contained within the MQDH structure. If these offsets result in unused bytes between one or more of the MQDH structure, the arrays of records, and the message data, those unused bytes must be included in the value of *StrucLength*, but the content of those bytes is not preserved by the queue manager. It is valid for the array of MQPMR records to precede the array of MQOR records.

The initial value of this field is 0.

Version (MQLONG)

The value must be:

MQDH_VERSION_1

Version number for distribution header structure.

The following constant specifies the version number of the current version:

MQDH_CURRENT_VERSION

Current version of distribution header structure.

The initial value of this field is MQDH_VERSION_1.

Initial values and language declarations for MQDH

Table 483. Initial values of fields in MQDH for MQDH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQDH_STRUC_ID	'DH¬¬'
<i>Version</i>	MQDH_VERSION_1	1
<i>StrucLength</i>	None	0
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQDHF_NONE	0
<i>PutMsgRecFields</i>	MQPMRF_NONE	0
<i>RecsPresent</i>	None	0
<i>ObjectRecOffset</i>	None	0
<i>PutMsgRecOffset</i>	None	0
Notes: <ol style="list-style-type: none"> 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQDH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQDH MyDH = {MQDH_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQDH MQDH;
struct tagMQDH {
    MQCHAR4  StrucId;           /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   StrucLength;      /* Length of MQDH structure plus following
                                MQOR and MQPMR records */
    MQLONG   Encoding;         /* Numeric encoding of data that follows
                                the MQOR and MQPMR records */
    MQLONG   CodedCharSetId;   /* Character set identifier of data that
                                follows the MQOR and MQPMR records */
    MQCHAR8  Format;           /* Format name of data that follows the
                                MQOR and MQPMR records */
    MQLONG   Flags;            /* General flags */
    MQLONG   PutMsgRecFields;  /* Flags indicating which MQPMR fields are
                                present */
    MQLONG   RecsPresent;      /* Number of MQOR records present */
}
```

```

    MQLONG    ObjectRecOffset; /* Offset of first MQOR record from start
                                of MQDH */
    MQLONG    PutMsgRecOffset; /* Offset of first MQPMR record from start
                                of MQDH */
};

```

COBOL declaration

```

**      MQDH structure
10 MQDH.
**      Structure identifier
15 MQDH-STRUCID      PIC X(4).
**      Structure version number
15 MQDH-VERSION      PIC S9(9) BINARY.
**      Length of MQDH structure plus following MQOR and MQPMR records
15 MQDH-STRUCLENGTH  PIC S9(9) BINARY.
**      Numeric encoding of data that follows the MQOR and MQPMR records
15 MQDH-ENCODING     PIC S9(9) BINARY.
**      Character set identifier of data that follows the MQOR and MQPMR
**      records
15 MQDH-CODEDCHARSETID PIC S9(9) BINARY.
**      Format name of data that follows the MQOR and MQPMR records
15 MQDH-FORMAT       PIC X(8).
**      General flags
15 MQDH-FLAGS        PIC S9(9) BINARY.
**      Flags indicating which MQPMR fields are present
15 MQDH-PUTMSGRECFIELDS PIC S9(9) BINARY.
**      Number of MQOR records present
15 MQDH-RECSPRESENT  PIC S9(9) BINARY.
**      Offset of first MQOR record from start of MQDH
15 MQDH-OBJECTRECOFFSET PIC S9(9) BINARY.
**      Offset of first MQPMR record from start of MQDH
15 MQDH-PUTMSGRECOFFSET PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQDH based,
3 StrucId      char(4),          /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 StrucLength  fixed bin(31), /* Length of MQDH structure plus
                                following MQOR and MQPMR
                                records */
3 Encoding      fixed bin(31), /* Numeric encoding of data that
                                follows the MQOR and MQPMR
                                records */
3 CodedCharSetId fixed bin(31), /* Character set identifier of data that
                                follows the MQOR and MQPMR
                                records */
3 Format        char(8),          /* Format name of data that follows
                                the MQOR and MQPMR records */
3 Flags        fixed bin(31), /* General flags */
3 PutMsgRecFields fixed bin(31), /* Flags indicating which MQPMR
                                fields are present */
3 RecsPresent  fixed bin(31), /* Number of MQOR records present */
3 ObjectRecOffset fixed bin(31), /* Offset of first MQOR record from
                                start of MQDH */
3 PutMsgRecOffset fixed bin(31); /* Offset of first MQPMR record from
                                start of MQDH */

```

Visual Basic declaration

```

Type MQDH
StrucId      As String*4 'Structure identifier'
Version      As Long     'Structure version number'
StrucLength  As Long     'Length of MQDH structure plus following'
                                'MQOR and MQPMR records'
Encoding     As Long     'Numeric encoding of data that follows'
                                'the MQOR and MQPMR records'
CodedCharSetId As Long   'Character set identifier of data that'
                                'follows the MQOR and MQPMR records'
Format       As String*8 'Format name of data that follows the'
                                'MQOR and MQPMR records'
Flags        As Long     'General flags'
PutMsgRecFields As Long   'Flags indicating which MQPMR fields are'
                                'present'

```

```

RecsPresent      As Long      'Number of MQOR records present'
ObjectRecOffset  As Long      'Offset of first MQOR record from start'
                                'of MQDH'
PutMsgRecOffset  As Long      'Offset of first MQPMR record from start'
                                'of MQDH'
End Type

```

MQDLH - Dead-letter header

The following table summarizes the fields in the structure.

Table 484. Fields in MQDLH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Reason</i>	Reason message arrived on dead-letter queue	Reason
<i>DestQName</i>	Name of original destination queue	DestQName
<i>DestQMgrName</i>	Name of original destination queue manager	DestQMgrName
<i>Encoding</i>	Numeric encoding of data that follows MQDLH	Encoding
<i>CodedCharSetId</i>	Character set identifier of data that follows MQDLH	CodedCharSetId
<i>Format</i>	Format name of data that follows MQDLH	Format
<i>PutApplType</i>	Type of application that put message on dead-letter queue	PutApplType
<i>PutApplName</i>	Name of application that put message on dead-letter queue	PutApplName
<i>PutDate</i>	Date when message was put on dead-letter queue	PutDate
<i>PutTime</i>	Time when message was put on dead-letter queue	PutTime

Overview for MQDLH

Availability: All IBM MQ platforms.

Purpose: The MQDLH structure describes the information that prefixes the application message data of messages on the dead-letter (undelivered-message) queue. A message can arrive on the dead-letter queue either because the queue manager or message channel agent has redirected it to the queue, or because an application has put the message directly on the queue.

Format name: MQFMT_DEAD_LETTER_HEADER.

Character set and encoding: The fields in the MQDLH structure are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields. These are specified in the header structure that precedes the MQDLH, or in the MQMD structure if the MQDLH is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

If you are using the WMQ classes for Java/JMS, and the code page defined in the MQMD is not supported by the Java virtual machine, then the MQDLH is written in the UTF-8 character set.

Usage: Applications that put messages directly on the dead-letter queue must prefix the message data with an MQDLH structure, and initialize the fields with appropriate values. However, the queue manager

does not require that an MQDLH structure be present, or that valid values have been specified for the fields.

If a message is too long to put on the dead-letter queue, the application must do one of the following:

- Truncate the message data to fit on the dead-letter queue.
- Record the message on auxiliary storage and place an exception report message on the dead-letter queue indicating this.
- Discard the message and return an error to its originator. If the message is (or might be) a critical message, do this only if it is known that the originator still has a copy of the message; for example, a message received by a message channel agent from a communication channel.

Which of the preceding actions is appropriate (if any) depends on the design of the application.

The queue manager performs special processing when a message that is a segment is put with an MQDLH structure at the front; see the description of the MQMDE structure for further details.

Putting messages on the dead-letter queue: When a message is put on the dead-letter queue, the MQMD structure used for the MQPUT or MQPUT1 call must be identical to the MQMD associated with the message (usually the MQMD returned by the MQGET call), with the exception of the following:

- Set the *CodedCharSetId* and *Encoding* fields to whatever character set and encoding are used for fields in the MQDLH structure.
- Set the *Format* field to MQFMT_DEAD_LETTER_HEADER to indicate that the data begins with a MQDLH structure.
- Set the context fields (*AccountingToken*, *ApplIdentityData*, *ApplOriginData*, *PutApplName*, *PutApplType*, *PutDate*, *PutTime*, *UserIdentifier*) by using a context option appropriate to the circumstances:
 - An application putting on the dead-letter queue a message that is not related to any preceding message must use the MQPMO_DEFAULT_CONTEXT option; this causes the queue manager to set all of the context fields in the message descriptor to their default values.
 - A server application putting on the dead-letter queue a message that it has just received must use the MQPMO_PASS_ALL_CONTEXT option to preserve the original context information.
 - A server application putting on the dead-letter queue a *reply* to a message that it has just received must use the MQPMO_PASS_IDENTITY_CONTEXT option; this preserves the identity information but sets the origin information to be that of the server application.
 - A message channel agent putting on the dead-letter queue a message that it received from its communication channel must use the MQPMO_SET_ALL_CONTEXT option to preserve the original context information.

In the MQDLH structure itself, set the fields as follows:

- Set the *CodedCharSetId*, *Encoding*, and *Format* fields to the values that describe the data that follows the MQDLH structure, usually the values from the original message descriptor.
- Set the context fields *PutApplType*, *PutApplName*, *PutDate*, and *PutTime* to values appropriate to the application that is putting the message on the dead-letter queue; these values are not related to the original message.
- Set other fields as appropriate.

Ensure that all fields have valid values, and that character fields are padded with blanks to the defined length of the field; do not end the character data prematurely by using a null character, because the queue manager does not convert the null and subsequent characters to blanks in the MQDLH structure.

Getting messages from the dead-letter queue: Applications that get messages from the dead-letter queue must verify that the messages begin with an MQDLH structure. The application can determine whether an MQDLH structure is present by examining the *Format* field in the message descriptor MQMD; if the field has the value MQFMT_DEAD_LETTER_HEADER, the message data begins with an MQDLH structure. Be aware also that messages that applications get from the dead-letter queue might be truncated if they were originally too long for the queue.

Fields for MQDLH

The MQDLH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

CodedCharSetId is the character set identifier of the data that flows through the MQDLH structure (usually the data from the original message); it does not apply to character data in the MQDLH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

MQCCSI_INHERIT

Character data in the data following this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

You cannot use MQCCSI_INHERIT if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

This value is supported in the following environments: AIX, HP-UX, z/OS, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

The initial value of this field is MQCCSI_UNDEFINED.

DestQMgrName (MQCHAR48)

DestQMgrName is the name of the queue manager that was the original destination for the message.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

DestQName (MQCHAR48)

DestQName is the name of the message queue that was the original destination for the message.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

Encoding (MQLONG)

Encoding is the numeric encoding of the data that follows the MQDLH structure (usually the data from the original message); it does not apply to numeric data in the MQDLH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

Format (MQCHAR8)

Format is the format name of the data that follows the MQDLH structure (usually the data from the original message).

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those rules for coding the *Format* field in MQMD.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

PutApplName (MQCHAR28)

PutApplName is the name of the application that put the message on the dead-letter (undelivered-message) queue.

The format of the name depends on the *PutApplType* field. The format can vary release to release. See the description of the *PutApplName* field in [“MQMD - Message descriptor”](#) on page 396.

If the queue manager redirects the message to the dead-letter queue, *PutApplName* contains the first 28 characters of the queue-manager name, padded with blanks if necessary.

The length of this field is given by MQ_PUT_APPL_NAME_LENGTH. The initial value of this field is the null string in C, and 28 blank characters in other programming languages.

PutApplType (MQLONG)

PutApplType is the type of application that put the message on the dead-letter (undelivered-message) queue.

This field has the same meaning as the *PutApplType* field in the message descriptor MQMD (see [“MQMD - Message descriptor”](#) on page 396 for details).

If the queue manager redirects the message to the dead-letter queue, *PutApplType* has the value MQAT_QMGR.

The initial value of this field is 0.

PutDate (MQCHAR8)

PutDate is the date when the message was put on the dead-letter (undelivered-message) queue.

The format used for the date when this field is generated by the queue manager is:

- YYYYMMDD

where the characters represent:

YYYY

year (four numeric digits)

MM

month of year (01 through 12)

DD

day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

The length of this field is given by MQ_PUT_DATE_LENGTH. The initial value of this field is the null string in C, and eight blank characters in other programming languages.

PutTime (MQCHAR8)

PutTime is the time when the message was put on the dead-letter (undelivered-message) queue.

The format used for the time when this field is generated by the queue manager is:

- HHMMSSTH

where the characters represent:

HH

hours (00 through 23)

MM

minutes (00 through 59)

SS

seconds (00 through 59; see note)

T

tenths of a second (0 through 9)

H

hundredths of a second (0 through 9)

Note: If the system clock is synchronized to an very accurate time standard, it is possible on rare occasions for 60 or 61 to be returned for the seconds in *PutTime*. This happens when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

The length of this field is given by MQ_PUT_TIME_LENGTH. The initial value of this field is the null string in C, and eight blank characters in other programming languages.

Reason (MQLONG)

The Reason field identifies the reason why the message was placed on the dead-letter queue instead of on the original destination queue.

This identifies the reason why the message was placed on the dead-letter queue instead of on the original destination queue. It should be one of the MQFB_* or MQRC_* values (for example, MQRC_Q_FULL). See the description of the *Feedback* field in [“MQMD - Message descriptor” on page 396](#) for details of the common MQFB_* values that can occur.

If the value is in the range MQFB_IMS_FIRST through MQFB_IMS_LAST, the actual IMS error code can be determined by subtracting MQFB_IMS_ERROR from the value of the *Reason* field.

Some MQFB_* values occur only in this field. They relate to repository messages, trigger messages, or transmission-queue messages that have been transferred to the dead-letter queue. These are:

MQFB_APPL_CANNOT_BE_STARTED (X'00000109')

An application processing a trigger message cannot start the application named in the *AppId* field of the trigger message (see [“MQTM - Trigger message” on page 572](#)).

On z/OS, the CKTI CICS transaction is an example of an application that processes trigger messages.

MQFB_APPL_TYPE_ERROR (X'0000010B')

An application processing a trigger message cannot start the application because the *AppType* field of the trigger message is not valid (see [“MQTM - Trigger message” on page 572](#)).

On z/OS, the CKTI CICS transaction is an example of an application that processes trigger messages.

MQFB_BIND_OPEN_CLUSRCVR_DEL (X'00000119')

The message was on the SYSTEM.CLUSTER.TRANSMIT.QUEUE intended for a cluster queue that was opened with the MQOO_BIND_ON_OPEN option, but the remote cluster-receiver channel to be used to transmit the message to the destination queue was deleted before the message could be sent. Because MQOO_BIND_ON_OPEN was specified, only the channel selected when the queue was opened can be used to transmit the message. As this channel is no longer available, the message is placed on the dead-letter queue.

MQFB_NOT_A_REPOSITORY_MSG (X'00000118')

The message is not a repository message.

MQFB_STOPPED_BY_CHAD_EXIT (X'00000115')

The message was stopped by channel auto-definition exit.

MQFB_STOPPED_BY_MSG_EXIT (X'0000010D')

The message was stopped by channel message exit.

MQFB_TM_ERROR (X'0000010A')

The *Format* field in MQMD specifies MQFMT_TRIGGER, but the message does not begin with a valid MQTM structure. For example, the *StrucId* mnemonic eye-catcher might not be valid, the *Version* might not be recognized, or the length of the trigger message might be insufficient to contain the MQTM structure.

On z/OS, the CKTI CICS transaction is an example of an application that processes trigger messages and can generate this feedback code.

MQFB_XMIT_Q_MSG_ERROR (X'0000010F')

A message channel agent has found that a message on the transmission queue is not in the correct format. The message channel agent puts the message on the dead-letter queue using this feedback code.

One common cause is that a message has been put directly to the transmission queue, so the message does not have the expected XQH header. Messages should be put to a transmission queue through a remote queue, unless the application builds the MQXQH header.

The initial value of this field is MQRC_NONE.

StrucId (MQCHAR4)

StrucId is the structure identifier.

The value must be:

MQDLH_STRUC_ID

Identifier for dead-letter header structure.

For the C programming language, the constant MQDLH_STRUC_ID_ARRAY is also defined; this has the same value as MQDLH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQDLH_STRUC_ID.

Version (MQLONG)

Version is the structure version number.

The value must be:

MQDLH_VERSION_1

Version number for dead-letter header structure.

The following constant specifies the version number of the current version:

MQDLH_CURRENT_VERSION

Current version of dead-letter header structure.

The initial value of this field is MQDLH_VERSION_1.

Initial values and language declarations for MQDLH

Table 485. Initial values of fields in MQDLH for MQDLH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQDLH_STRUC_ID	'DLH~'
<i>Version</i>	MQDLH_VERSION_1	1
<i>Reason</i>	MQRC_NONE	0
<i>DestQName</i>	None	Null string or blanks
<i>DestQMgrName</i>	None	Null string or blanks
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>PutApplType</i>	None	0
<i>PutApplName</i>	None	Null string or blanks
<i>PutDate</i>	None	Null string or blanks
<i>PutTime</i>	None	Null string or blanks

Table 485. Initial values of fields in MQDLH for MQDLH (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQDLH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQDLH MyDLH = {MQDLH_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQDLH MQDLH;
struct tagMQDLH {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;          /* Structure version number */
    MQLONG    Reason;           /* Reason message arrived on dead-letter
                                (undelivered-message) queue */
    MQCHAR48  DestQName;        /* Name of original destination queue */
    MQCHAR48  DestQMgrName;     /* Name of original destination queue
                                manager */
    MQLONG    Encoding;         /* Numeric encoding of data that follows
                                MQDLH */
    MQLONG    CodedCharSetId;   /* Character set identifier of data that
                                follows MQDLH */
    MQCHAR8   Format;           /* Format name of data that follows
                                MQDLH */
    MQLONG    PutApplType;      /* Type of application that put message on
                                dead-letter (undelivered-message)
                                queue */
    MQCHAR28  PutApplName;      /* Name of application that put message on
                                dead-letter (undelivered-message)
                                queue */
    MQCHAR8   PutDate;          /* Date when message was put on dead-letter
                                (undelivered-message) queue */
    MQCHAR8   PutTime;          /* Time when message was put on the
                                dead-letter (undelivered-message)
                                queue */
};
```

COBOL declaration

```
** MQDLH structure
10 MQDLH.
** Structure identifier
15 MQDLH-STRUCID PIC X(4).
** Structure version number
15 MQDLH-VERSION PIC S9(9) BINARY.
** Reason message arrived on dead-letter (undelivered-message) queue
15 MQDLH-REASON PIC S9(9) BINARY.
** Name of original destination queue
15 MQDLH-DESTQNAME PIC X(48).
** Name of original destination queue manager
15 MQDLH-DESTQMGRNAME PIC X(48).
** Numeric encoding of data that follows MQDLH
15 MQDLH-ENCODING PIC S9(9) BINARY.
** Character set identifier of data that follows MQDLH
15 MQDLH-CODEDCHARSETID PIC S9(9) BINARY.
** Format name of data that follows MQDLH
15 MQDLH-FORMAT PIC X(8).
** Type of application that put message on dead-letter
** (undelivered-message) queue
15 MQDLH-PUTAPPLTYPE PIC S9(9) BINARY.
** Name of application that put message on dead-letter
** (undelivered-message) queue
15 MQDLH-PUTAPPLNAME PIC X(28).
** Date when message was put on dead-letter (undelivered-message)
** queue
15 MQDLH-PUTDATE PIC X(8).
```

```

**      Time when message was put on the dead-letter (undelivered-message)
**      queue
15 MQDLH-PUTTIME          PIC X(8).

```

PL/I declaration

```

dcl
  1 MQDLH based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 Reason       fixed bin(31), /* Reason message arrived on
                                   dead-letter (undelivered-message)
                                   queue */
    3 DestQName    char(48),     /* Name of original destination
                                   queue */
    3 DestQMgrName char(48),     /* Name of original destination queue
                                   manager */
    3 Encoding     fixed bin(31), /* Numeric encoding of data that
                                   follows MQDLH */
    3 CodedCharSetId fixed bin(31), /* Character set identifier of data
                                   that follows MQDLH */
    3 Format        char(8),      /* Format name of data that follows
                                   MQDLH */
    3 PutApplType   fixed bin(31), /* Type of application that put
                                   message on dead-letter
                                   (undelivered-message) queue */
    3 PutApplName   char(28),     /* Name of application that put
                                   message on dead-letter
                                   (undelivered-message) queue */
    3 PutDate       char(8),      /* Date when message was put on
                                   dead-letter (undelivered-message)
                                   queue */
    3 PutTime       char(8);      /* Time when message was put on the
                                   dead-letter (undelivered-message)
                                   queue */

```

High Level Assembler declaration

MQDLH	DSECT	
MQDLH_STRUCID	DS	CL4 Structure identifier
MQDLH_VERSION	DS	F Structure version number
MQDLH_REASON	DS	F Reason message arrived on dead-letter
*		(undelivered-message) queue
MQDLH_DESTQNAME	DS	CL48 Name of original destination queue
MQDLH_DESTQMGRNAME	DS	CL48 Name of original destination queue
*		manager
MQDLH_ENCODING	DS	F Numeric encoding of data that follows
*		MQDLH
MQDLH_CODEDCHARSETID	DS	F Character set identifier of data that
*		follows MQDLH
MQDLH_FORMAT	DS	CL8 Format name of data that follows MQDLH
MQDLH_PUTAPPLTYPE	DS	F Type of application that put message on
*		dead-letter (undelivered-message) queue
MQDLH_PUTAPPLNAME	DS	CL28 Name of application that put message on
*		dead-letter (undelivered-message) queue
MQDLH_PUTDATE	DS	CL8 Date when message was put on
*		dead-letter (undelivered-message) queue
MQDLH_PUTTIME	DS	CL8 Time when message was put on the
*		dead-letter (undelivered-message) queue
*		
MQDLH_LENGTH	EQU	*-MQDLH
	ORG	MQDLH
MQDLH_AREA	DS	CL(MQDLH_LENGTH)

Visual Basic declaration

Type MQDLH		
StrucId	As String*4	'Structure identifier'
Version	As Long	'Structure version number'
Reason	As Long	'Reason message arrived on dead-letter'
		'(undelivered-message) queue'
DestQName	As String*48	'Name of original destination queue'
DestQMgrName	As String*48	'Name of original destination queue'
		'manager'
Encoding	As Long	'Numeric encoding of data that follows'
		'MQDLH'

CodedCharSetId	As Long	'Character set identifier of data that follows MQDLH'
Format	As String*8	'Format name of data that follows MQDLH'
PutApplType	As Long	'Type of application that put message on dead-letter (undelivered-message) queue'
PutApplName	As String*28	'Name of application that put message on dead-letter (undelivered-message) queue'
PutDate	As String*8	'Date when message was put on dead-letter (undelivered-message) queue'
PutTime	As String*8	'Time when message was put on the dead-letter (undelivered-message) queue'
End	Type	

MQDMHO - Delete message handle options

The following table summarizes the fields in the structure.

Table 486. Fields in MQDMHO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options

Overview for MQDMHO

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The **MQDMHO** structure allows applications to specify options that control how message handles are deleted. The structure is an input parameter on the **MQDLTMH** call.

Character set and encoding: Data in **MQDMHO** must be in the character set of the application and encoding of the application (**MQENC_NATIVE**).

Fields for MQDMHO

The MQDMHO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

The value must be:

MQDMHO_NONE

No options specified.

This is always an input field. The initial value of this field is **MQDMHO_NONE**.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQDMHO_STRUC_ID

Identifier for delete message handle options structure.

For the C programming language, the constant **MQDMHO_STRUC_ID_ARRAY** is also defined; this has the same value as **MQDMHO_STRUC_ID**, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is **MQDMHO_STRUC_ID**.

Version (MQLONG)

This is the structure version number; the value must be:

MQDMHO_VERSION_1

Version-1 delete message handle options structure.

The following constant specifies the version number of the current version:

MQDMHO_CURRENT_VERSION

Current version of delete message handle options structure.

This is always an input field. The initial value of this field is **MQDMHO_VERSION_1**.

Initial values and language declarations for MQDMHO

Table 487. Initial values of fields in MQDMHO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQDMHO_STRUC_ID	' DMHO '
<i>Version</i>	MQDMHO_VERSION_1	1
<i>Options</i>	MQDMHO_NONE	0
Notes: 1. In the C programming language, the macro variable MQDMHO_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQDMHO MyDMHO = {MQDMHO_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQDMHO;  
struct tagMQDMHO {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    MQLONG     Options;          /* Options that control the action of MQDLTMH */  
};
```

COBOL declaration

```
**    MQDMHO structure  
10 MQDMHO.  
**    Structure identifier  
15 MQDMHO-STRUCID      PIC X(4).  
**    Structure version number  
15 MQDMHO-VERSION      PIC S9(9) BINARY.  
**    Options that control the action of MQDLTMH  
15 MQDMHO-OPTIONS      PIC S9(9) BINARY.
```

PL/I declaration

```
dcl  
1 MQDMHO based,  
3 StrucId      char(4),          /* Structure identifier */  
3 Version      fixed bin(31),    /* Structure version number */  
3 Options      fixed bin(31),    /* Options that control the action of MQDLTMH */
```

High Level Assembler declaration

```
MQDMHO          DSECT  
MQDMHO_STRUCID  DS    CL4      Structure identifier  
MQDMHO_VERSION  DS    F        Structure version number  
MQDMHO_OPTIONS  DS    F        Options that control the action of  
*                      MQDLTMH
```

MQDMHO_LENGTH	EQU	*-MQDMHO
MQDMHO_AREA	DS	CL(MQDMHO_LENGTH)

MQDMPO - Delete message property options

The following table summarizes the fields in the structure. MQDMPO structure - delete message property options

Table 488. Fields in MQDMPO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options controlling the action of MQDMPO	Options

Overview for MQDMPO

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The MQDMPO structure allows applications to specify options that control how properties of messages are deleted. The structure is an input parameter on the MQDLTMP call.

Character set and encoding: Data in MQDMPO must be in the character set of the application and encoding of the application (MQENC_NATIVE).

Fields for MQDMPO

Delete message property options structure - fields

The MQDMPO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

Delete message property options structure - Options field

Location options: The following options relate to the relative location of the property compared to the property cursor.

MQDMPO_DEL_FIRST

Deletes the first property that matches the specified name.

MQDMPO_DEL_PROP_UNDER_CURSOR

Deletes the property pointed to by the property cursor; that is the property that was last inquired by using either the MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused. It is also reset when the message handle is specified in the *MsgHandle* field of the MQGMO structure on an MQGET call, or MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established, the call fails with completion code MQCC_FAILED and reason MQRC_PROPERTY_NOT_AVAILABLE. If the property pointed to by the property cursor has already been deleted, the call also fails with completion code MQCC_FAILED and reason MQRC_PROPERTY_NOT_AVAILABLE.

If neither of the options is required, the following option can be used:

MQDMPO_NONE

No options specified.

This field is always an input field. The initial value of this field is MQDMPO_DEL_FIRST.

StrucId (MQCHAR4)

Delete message property options structure - StrucId field

This is the structure identifier. The value must be:

MQDMPO_STRUC_ID

Identifier for delete message property options structure.

For the C programming language, the constant MQDMPO_STRUC_ID_ARRAY is also defined; this has the same value as MQDMPO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQDMPO_STRUC_ID.

Version (MQLONG)

Delete message property options structure - Version field

This is the structure version number. The value must be:

MQDMPO_VERSION_1

Version number for delete message property options structure.

The following constant specifies the version number of the current version:

MQDMPO_CURRENT_VERSION

Current version of delete message property options structure.

This is always an input field. The initial value of this field is MQDMPO_VERSION_1.

Initial values and language declarations for MQDMPO

Delete message property options structure - Initial values

Table 489. Initial values of fields in MQDPMO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQDMPO_STRUC_ID	' DMP0 '
<i>Version</i>	MQDMPO_VERSION_1	1
<i>Options</i>	Options that control the action of MQDLTMP	MQDMPO_NONE
Notes: 1. In the C programming language, the macro variable MQDMPO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQDMPO MyDMP0 = {MQDMPO_DEFAULT};</pre>		

C declaration

Delete message property options structure - C language declaration

```
typedef struct tagMQDMPO MQDMPO;
struct tagMQDMPO {
    MQCHAR4    StrucId;        /* Structure identifier */
    MQLONG     Version;        /* Structure version number */
    MQLONG     Options;        /* Options that control the action of
                               MQDLTMP */
};
```

COBOL declaration

Delete message property options structure - COBOL language declaration

```
**      MQDMPO structure
10      MQDMPO.
**      Structure identifier
15      MQDMPO-STRUCID          PIC X(4).
**      Structure version number
15      MQDMPO-VERSION          PIC S9(9) BINARY.
```

```

**      Options that control the action of MQDLTMP
15 MQDPMO-OPTIONS          PIC S9(9) BINARY.

```

PL/I declaration

Delete message property options structure - PL/I language declaration

```

Dcl
  1 MQDPMO based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 Options      fixed bin(31), /* Options that control the action
                                   of MQDLTMP */

```

High Level Assembler declaration

Delete message property options structure - Assembler language declaration

```

MQDPMO          DSECT
MQDPMO_STRUCID  DS   CL4  Structure identifier
MQDPMO_VERSION  DS   F    Structure version number
MQDPMO_OPTIONS  DS   F    Options that control the
*                  action of MQDLTMP
MQDPMO_LENGTH   EQU   *-MQDPMO
MQDPMO_AREA     DS    CL(MQDPMO_LENGTH)

```

MQEPH - Embedded PCF header

The following table summarizes the fields in the structure.

Table 490. Fields in MQEPH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQEPH structure plus the MQCFH and parameter structures that follow it	StrucLength
<i>Encoding</i>	Numeric encoding of data that follows last PCF parameter structure	Encoding
<i>CodedCharSetId</i>	Character set identifier of data that follows last PCF parameter structure	CodedCharSetId
<i>Format</i>	Format name of data that follows last PCF parameter structure	Format
<i>Flags</i>	Flags	Flags
<i>PCFHeader</i>	Programmable command format (PCF) header	PCFHeader

Overview for MQEPH

Availability: All IBM MQ platforms.

Purpose: The MQEPH structure describes the additional data that is present in a message when that message is a programmable command format (PCF) message. The *PCFHeader* field defines the PCF parameters that follow this structure and this allows you to follow the PCF message data with other headers.

Format name: MQFMT_EMBEDDED_PCF

Character set and encoding: Data in MQEPH must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE.

Set the character set and encoding of the MQEPH into the *CodedCharSetId* and *Encoding* fields in:

- The MQMD (if the MQEPH structure is at the start of the message data), or
- The header structure that precedes the MQEPH structure (all other cases).

Usage: You cannot use MQEPH structures to send commands to the command server or any other queue manager PCF-accepting server.

Similarly, the command server or any other queue manager PCF-accepting server do not generate responses or events containing MQEPH structures.

Fields for MQEPH

The MQEPH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This is the character set identifier of the data that follows the MQEPH structure and the associated PCF parameters; it does not apply to character data in the MQEPH structure itself.

The initial value of this field is MQCCSI_UNDEFINED.

Encoding (MQLONG)

This is the numeric encoding of the data that follows the MQEPH structure and the associated PCF parameters; it does not apply to character data in the MQEPH structure itself.

The initial value of this field is 0.

Flags (MQLONG)

The following values are available:

MQEPH_NONE

No flags have been specified. MQEPH_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

MQEPH_CCSID_EMBEDDED

The character set of the parameters containing character data is specified individually within the CodedCharSetId field in each structure. The character set of the StrucId and Format fields is defined by the CodedCharSetId field in the header structure that precedes the MQEPH structure, or by the CodedCharSetId field in the MQMD if the MQEPH is at the start of the message.

The initial value of this field is MQEPH_NONE.

Format (MQCHAR8)

This is the format name of the data that follows the MQEPH structure and the associated PCF parameters.

The initial value of this field is MQFMT_NONE.

PCFHeader (MQCFH)

This is the programmable command format (PCF) header, defining the PCF parameters that follow the MQEPH structure. This enables you to follow the PCF message data with other headers.

The PCF header is initially defined with the following values:

<i>Table 491. Initial values of fields in MQCFH</i>		
Field name	Name of constant	Value of constant
<i>Type</i>	MQCFT_NONE	0
<i>StrucLength</i>	MQCFH_STRUC_LENGTH	36
<i>Version</i>	MQCFH_VERSION_3	3
<i>StrucLength</i>	None	0

Table 491. Initial values of fields in MQCFH (continued)		
Field name	Name of constant	Value of constant
<i>Command</i>	MQCMD_NONE	0
<i>MsgSeqNumber</i>	None	1
<i>Control</i>	MQCFC_LAST	1
<i>CompCode</i>	MQCC_OK	0
<i>Reason</i>	MQRC_NONE	0
<i>ParameterCount</i>	None	0

The application must change the Type from MQCFT_NONE to a valid structure type for the use it is making of the embedded PCF header.

StrucId (MQCHAR4)

The value must be:

MQEPH_STRUC_ID

Identifier for distribution header structure.

For the C programming language, the constant MQEPH_STRUC_ID_ARRAY is also defined; this has the same value as MQDH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQEPH_STRUC_ID.

StrucLength (MQLONG)

This is the amount of data preceding the next header structure. It includes:

- The length of the MQEPH header
- The length of all PCF parameters following the header
- Any blank padding following those parameters

StrucLength must be a multiple of 4.

The fixed length part of the structure is defined by MQEPH_STRUC_LENGTH_FIXED.

The initial value of this field is 68.

Version (MQLONG)

The value must be:

MQEPH_VERSION_1

Version number for embedded PCF header structure.

The following constant specifies the version number of the current version:

MQCFH_VERSION_3

Current version of embedded PCF header structure.

The initial value of this field is MQEPH_VERSION_1.

Initial values and language declarations for MQEPH

Table 492. Initial values of fields in MQEPH for MQEPH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQEPH_STRUC_ID	'EPH↵'
<i>Version</i>	MQEPH_VERSION_1	1

Table 492. Initial values of fields in MQEPH for MQEPH (continued)

Field name	Name of constant	Value of constant
<i>StrucLength</i>	MQEPH_STRUC_LENGTH_FIXED	68
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQEPH_NONE	0
<i>PCFHeader</i>	Names and values as defined in Table 491 on page 346	0

Notes:

1. The symbol ~ represents a single blank character.
2. In the C programming language, the macro variable MQEPH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:

```
MQEPH MyEPH = {MQEPH_DEFAULT};
```

C declaration

```
typedef struct tagMQEPH MQEPH;
struct tagMQDH {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;          /* Structure version number */
    MQLONG    StrucLength;      /* Total length of MQEPH including the MQCFH
                                and parameter structures that follow it */
    MQLONG    Encoding;         /* Numeric encoding of data that follows last
                                PCF parameter structure */
    MQLONG    CodedCharSetId;   /* Character set identifier of data that
                                follows last PCF parameter structure */
    MQCHAR8   Format;           /* Format name of data that follows last PCF
                                parameter structure */
    MQLONG    Flags;            /* Flags */
    MQCFH     PCFHeader;        /* Programmable command format header */
};
```

COBOL declaration

```
**  MQEPH structure
10 MQEPH.
**    Structure identifier
15 MQEPH-STRUCID          PIC X(4).
**    Structure version number
15 MQEPH-VERSION          PIC S9(9) BINARY.
**    Total length of MQEPH structure including the MQCFH
**    and parameter structures that follow it
15 MQEPH-STRUCLength     PIC S9(9) BINARY.
**    Numeric encoding of data that follows last
**    PCF structure
15 MQEPH-ENCODING         PIC S9(9) BINARY.
**    Character set identifier of data that
**    follows last PCF parameter structure
15 MQEPH-CODEDCHARSETID  PIC S9(9) BINARY.
**    Format name of data that follows last PCF
**    parameter structure
15 MQEPH-FORMAT           PIC X(8).
**    Flags
15 MQEPH-FLAGS            PIC S9(9) BINARY.
**    Programmable command format header
15 MQEPH-PCFHEADER.
**    Structure type
20 MQEPH-PCFHEADER-TYPE   PIC S9(9) BINARY.
**    Structure length
20 MQEPH-PCFHEADER-STRUCLength PIC S9(9) BINARY.
**    Structure version number
```

```

20 MQEPH-PCFHEADER-VERSION      PIC S9(9) BINARY.
** Command identifier
20 MQEPH-PCFHEADER-COMMAND      PIC S9(9) BINARY.
** Message sequence number
20 MQEPH-PCFHEADER-MSGSEQNUMBER PIC S9(9) BINARY.
** Control options
20 MQEPH-PCFHEADER-CONTROL      PIC S9(9) BINARY.
** Completion code
20 MQEPH-PCFHEADER-COMPCODE     PIC S9(9) BINARY.
** Reason code qualifying completion code
20 MQEPH-PCFHEADER-REASON      PIC S9(9) BINARY.
** Count of parameter structures
20 MQEPH-PCFHEADER-PARAMETERCOUNT PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQEPH based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 StrucLength  fixed bin(31), /* Total Length of MQEPH including the
                                   MQCFH and parameter structures that
                                   follow it
    3 Encoding     fixed bin(31), /* Numeric encoding of data that follows
                                   last PCF parameter structure
    3 CodedCharSetId fixed bin(31), /* Character set identifier of data that
                                   follows last PCF parameter structure
    3 Format        char(8),      /* Format name of data that follows last
                                   PCF parameter structure */
    3 Flags        fixed bin(31), /* Flags */
    3 PCFHeader,   /* Programmable command format header
    5 Type         fixed bin(31), /* Structure type */
    5 StrucLength  fixed bin(31), /* Structure length */
    5 Version      fixed bin(31), /* Structure version number */
    5 Command      fixed bin(31), /* Command identifier */
    5 MsgseqNumber fixed bin(31), /* Message sequence number */
    5 Control      fixed bin(31), /* Control options */
    5 CompCode     fixed bin(31), /* Completion code */
    5 Reason       fixed bin(31), /* Reason code qualifying completion code */
    5 ParameterCount fixed bin(31); /* Count of parameter structures */

```

High Level Assembler declaration

```

MQEPH          DSECT
MQEPH_STRUCID  DS CL4  Structure identifier
MQEPH_VERSION  DS F    Structure version number
MQEPH_STRUCLNGTH DS F    Total length of MQEPH including the
*              MQCFH and parameter structures that
*              follow it
MQEPH_ENCODING DS F    Numeric encoding of data that follows
*              last PCF parameter structure
MQEPH_CODEDCHARSETID DS F    Character set identifier of data that
*              follows last PCF parameter structure
MQEPH_FORMAT   DS CL8  Format name of data that follows last
*              PCF parameter structure
MQEPH_FLAGS    DS F    Flags
MQEPH_PCFHEADER DS 0F   Force fullword alignment
MQEPH_PCFHEADER_TYPE DS F    Structure type
MQEPH_PCFHEADER_STRUCLNGTH DS F    Structure length
MQEPH_PCFHEADER_VERSION DS F    Structure version number
MQEPH_PCFHEADER_COMMAND DS F    Command identifier
MQEPH_PCFHEADER_MSGSEQNUMBER DS F    Structure length
MQEPH_PCFHEADER_CONTROL DS F    Control options
MQEPH_PCFHEADER_COMPCODE DS F    Completion code
MQEPH_PCFHEADER_REASON DS F    Reason code qualifying completion code
MQEPH_PCFHEADER_PARAMETER COUNT DS F    Count of parameter structures
MQEPH_PCFHEADER_LENGTH EQU *-MQEPH_PCFHEADER
ORG MQEPH_PCFHEADER
MQEPH_PCFHEADER_AREA DS CL(MQEPH_PCFHEADER_LENGTH)
*
MQEPH_LENGTH EQU *-MQEPH
ORG MQEPH
MQEPH_AREA DS CL(MQEPH_LENGTH)

```

Visual Basic declaration

```

Type MQEPH
  StrucId      As String*4 'Structure identifier'
  Version      As Long     'Structure version number'
  StrucLength  As Long     'Total length of MQEPH structure including the MQCFH'
                                'and parameter structures that follow it'
  Encoding     As Long     'Numeric encoding of data that follows last'
                                'PCF parameter structure'
  CodedCharSetId As Long   'Character set identifier of data that'
                                'follows last PCF parameter structure'
  Format       As String*8 'Format name of data that follows last PCF'
                                'parameter structure'
  Flags        As Long     'Flags'
  PCFHeader    As MQCFH    'Programmable command format header'
End Type

Global MQEPH_DEFAULT As MQEPH

```

MQGMO - Get-message options

The following table summarizes the fields in the structure.

Table 493. Fields in MQGMO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options that control the action of MQGET	MQGMO - Options field
<i>WaitInterval</i>	Wait interval	WaitInterval
<i>Signal1</i>	Signal	Signal1
<i>Signal2</i>	Signal identifier	Signal2
<i>ResolvedQName</i>	Resolved name of destination queue	ResolvedQName
Note: The remaining fields are ignored if <i>Version</i> is less than MQGMO_VERSION_2.		
<i>MatchOptions</i>	Options controlling selection criteria used for MQGET	MatchOptions
<i>GroupStatus</i>	Flag indicating whether message retrieved is in a group	GroupStatus
<i>SegmentStatus</i>	Flag indicating whether message retrieved is a segment of a logical message	SegmentStatus
<i>Segmentation</i>	Flag indicating whether further segmentation is allowed for the message retrieved	Segmentation
<i>Reserved1</i>	Reserved	Reserved1
Note: The remaining fields are ignored if <i>Version</i> is less than MQGMO_VERSION_3.		
<i>MsgToken</i>	Message token	MsgToken
<i>ReturnedLength</i>	Length of message data returned (bytes)	ReturnedLength
Note: The remaining fields are ignored if <i>Version</i> is less than MQGMO_VERSION_4.		
<i>Reserved2</i>	Reserved	Reserved2

Table 493. Fields in MQGMO (continued)

Field	Description	Topic
<i>MsgHandle</i>	The handle to a message that is to be populated with the properties of the message being retrieved from the queue.	MsgHandle

Overview for MQGMO

Availability: All IBM MQ platforms.

Purpose: The MQGMO structure allows the application to control how messages are removed from queues. The structure is an input/output parameter on the MQGET call.

Version: The current version of MQGMO is MQGMO_VERSION_4. Certain fields are available only in certain versions of MQGMO. If you need to port applications between several environments, you must ensure that the version of MQGMO is consistent across all environments. Fields that exist only in particular versions of the structure are identified as such in [“MQGMO - Get-message options” on page 350](#) and in the field descriptions.

The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQGMO that is supported by the environment, but with the initial value of the *Version* field set to MQGMO_VERSION_1. To use fields that are not present in the version-1 structure, set the *Version* field to the version number of the version required.

Character set and encoding: Data in MQGMO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQGMO

The MQGMO structure contains the following fields; the fields are described in **alphabetical order**:

GroupStatus (MQCHAR)

This flag indicates whether the message retrieved is in a group.

It has one of the following values:

MQGS_NOT_IN_GROUP

Message is not in a group.

MQGS_MSG_IN_GROUP

Message is in a group, but is not the last in the group.

MQGS_LAST_MSG_IN_GROUP

Message is the last in the group.

This is also the value returned if the group consists of only one message.

This is an output field. The initial value of this field is MQGS_NOT_IN_GROUP. This field is ignored if *Version* is less than MQGMO_VERSION_2.

MatchOptions (MQLONG)

These options allow the application to choose which fields in the *MsgDesc* parameter to use to select the message returned by the MQGET call. The application sets the required options in this field, and then sets the corresponding fields in the *MsgDesc* parameter to the values required for those fields. Only messages that have those values in the MQMD for the message are candidates for retrieval using that *MsgDesc* parameter on the MQGET call. Fields for which the corresponding match option is *not* specified are ignored when selecting the message to be returned. If you specify no selection criteria on the MQGET call (that is, *any* message is acceptable), set *MatchOptions* to MQMO_NONE.

- On z/OS, the selection criteria that can be used might be restricted by the type of index used for the queue. See the *IndexType* queue attribute for further details.

If you specify MQGMO_LOGICAL_ORDER, only certain messages are eligible for return by the next MQGET call:

- If there is no current group or logical message, only messages that have *MsgSeqNumber* equal to 1 and *Offset* equal to 0 are eligible for return. In this situation, you can use one or more of the following match options to select which of the eligible messages is returned:
 - MQMO_MATCH_MSG_ID
 - MQMO_MATCH_CORREL_ID
 - MQMO_MATCH_GROUP_ID
- If there is a current group or logical message, only the next message in the group or next segment in the logical message is eligible for return, and this cannot be altered by specifying MQMO_* options.

In both of the preceding cases, you can specify match options that do not apply, but the value of the relevant field in the *MsgDesc* parameter must match the value of the corresponding field in the message to be returned; the call fails with reason code MQRC_MATCH_OPTIONS_ERROR if this condition is not satisfied.

MatchOptions is ignored if you specify either MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR.

Getting messages based on message property is not done using match options; for more information, see [“SelectionString \(MQCHARV\)” on page 464](#).

You can specify one or more of the following match options:

MQMO_MATCH_MSG_ID

The message to be retrieved must have a message identifier that matches the value of the *MsgId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the correlation identifier).

If you omit this option, the *MsgId* field in the *MsgDesc* parameter is ignored, and any message identifier will match.

Note: The message identifier MQMI_NONE is a special value that matches *any* message identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_MSG_ID with MQMI_NONE is the same as *not* specifying MQMO_MATCH_MSG_ID.

MQMO_MATCH_CORREL_ID

The message to be retrieved must have a correlation identifier that matches the value of the *CorrelId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the message identifier).

If you omit this option, the *CorrelId* field in the *MsgDesc* parameter is ignored, and any correlation identifier will match.

Note: The correlation identifier MQCI_NONE is a special value that matches *any* correlation identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_CORREL_ID with MQCI_NONE is the same as *not* specifying MQMO_MATCH_CORREL_ID.

MQMO_MATCH_GROUP_ID

The message to be retrieved must have a group identifier that matches the value of the *GroupId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the correlation identifier).

If you omit this option, the *GroupId* field in the *MsgDesc* parameter is ignored, and any group identifier will match.

Note: The group identifier MQGI_NONE is a special value that matches *any* group identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_GROUP_ID with MQGI_NONE is the same as *not* specifying MQMO_MATCH_GROUP_ID.

MQMO_MATCH_MSG_SEQ_NUMBER

The message to be retrieved must have a message sequence number that matches the value of the *MsgSeqNumber* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the group identifier).

If you omit this option, the *MsgSeqNumber* field in the *MsgDesc* parameter is ignored, and any message sequence number will match.

MQMO_MATCH_OFFSET

The message to be retrieved must have an offset that matches the value of the *Offset* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the message sequence number).

If you omit this option is not specified, the *Offset* field in the *MsgDesc* parameter is ignored, and any offset will match.

- This option is not supported on z/OS.

MQMO_MATCH_MSG_TOKEN

The message to be retrieved must have a message token that matches the value of the *MsgToken* field in the MQGMO structure specified on the MQGET call.

You can specify this option for all local queues. If you specify it for a queue that has an *IndexType* of MQIT_MSG_TOKEN (a WLM-managed queue), you can specify no other match options with MQMO_MATCH_MSG_TOKEN.

You cannot specify MQMO_MATCH_MSG_TOKEN with MQGMO_WAIT or MQGMO_SET_SIGNAL. If the application wants to wait for a message to arrive on a queue that has an *IndexType* of MQIT_MSG_TOKEN, specify MQMO_NONE.

If you omit this option, the *MsgToken* field in MQGMO is ignored, and any message token will match.

If you specify none of the options described, you can use the following option:

MQMO_NONE

Use no matches in selecting the message to be returned; all messages on the queue are eligible for retrieval (but subject to control by the MQGMO_ALL_MSGS_AVAILABLE, MQGMO_ALL_SEGMENTS_AVAILABLE, and MQGMO_COMPLETE_MSG options).

MQMO_NONE aids program documentation. It is not intended that this option be used with any other MQMO_* option, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of this field is MQMO_MATCH_MSG_ID with MQMO_MATCH_CORREL_ID. This field is ignored if *Version* is less than MQGMO_VERSION_2.

Note: The initial value of the *MatchOptions* field is defined for compatibility with earlier MQSeries® queue managers. However, when reading a series of messages from a queue without using selection criteria, this initial value requires the application to reset the *MsgId* and *CorrelId* fields to MQMI_NONE and MQCI_NONE before each MQGET call. Avoid the need to reset *MsgId* and *CorrelId* by setting *Version* to MQGMO_VERSION_2, and *MatchOptions* to MQMO_NONE.

Related concepts

[Message selectors in JMS](#)

MsgHandle (MQHMSG)

If the MQGMO_PROPERTIES_AS_Q_DEF option is specified and the PropertyControl queue attribute is not set to MQPROP_FORCE_MQRFH2 then this is the handle to a message which will be populated with the properties of the message being retrieved from the queue. The handle is created by an MQCRTMH call. Any properties already associated with the handle will be cleared before retrieving a message.

The following value can also be specified:

MQHM_NONE

No message handle supplied.

No message descriptor is required on the MQGET call if a valid message handle is supplied and used on output to contain the message properties, the message descriptor associated with the message handle is used for input fields.

If a message descriptor is specified on the MQGET call, it always takes precedence over the message descriptor associated with a message handle.

If MQGMO_PROPERTIES_FORCE_MQRFH2 is specified, or the MQGMO_PROPERTIES_AS_Q_DEF is specified and the PropertyControl queue attribute is MQPROP_FORCE_MQRFH2 then the call fails with reason code MQRC_MD_ERROR when no message descriptor parameter is specified.

On return from the MQGET call, the properties and message descriptor associated with this message handle are updated to reflect the state of the message retrieved (as well as the message descriptor if one was supplied on the MQGET call). The properties of the message can then be inquired using the MQINQMP call.

Except for message descriptor extensions, when present, a property that can be inquired with the MQINQMP call is not contained in the message data; if the message on the queue contained properties in the message data these are removed from the message data before the data is returned to the application.

If no message handle is provided or Version is less than MQGMO_VERSION_4 then you must supply a valid message descriptor on the MQGET call. Any message properties (except those contained in the message descriptor) are returned in the message data subject to the value of the property options in the MQGMO structure and the PropertyControl queue attribute.

This is an always an input field. The initial value of this field is MQHM_NONE. This field is ignored if Version is less than MQGMO_VERSION_4.

MsgToken (MQBYTE16)

MsgToken field - MQGMO structure. This field is used by the queue manager to uniquely identify a message.

This is a byte string that is generated by the queue manager to identify a message uniquely on a queue. The message token is generated when the message is first placed on the queue manager, and remains with the message until the message is permanently removed from the queue manager, unless the queue manager is restarted.

When the message is removed from the queue, the *MsgToken* that identified that instance of the message is no longer valid, and is never reused. If the queue manager is restarted, the *MsgToken* that identified a message on the queue before restart might not be valid after restart. However, the *MsgToken* is never reused to identify a different message instance. The *MsgToken* is generated by the queue manager and is not visible to any external application.

When a message is returned by a call to MQGET where a Version 3 or higher MQGMO is supplied, the *MsgToken* identifying the message on the queue is returned in the MQGMO by the queue manager. There is one exception to this: when the message is being removed from the queue outside syncpoint, the queue manager might not return a *MsgToken* because it is not useful to identify the returned message on a subsequent MQGET call. Applications should only use *MsgToken* to refer to the message on subsequent MQGET calls.

If a *MsgToken* is supplied and the *MatchOption* MQMO_MATCH_MSG_TOKEN is specified and neither MQGMO_MSG_UNDER_CURSOR nor MQGMO_BROWSE_MSG_UNDER_CURSOR is specified, only the message identified by that *MsgToken* can be returned. The option is valid on all local queues regardless of INDXTYPE, and on z/OS you must use INDXTYPE(MSGTOKEN) only on Workload Manager (WLM) queues.

Any other *MatchOptions* specified are checked, and if they do not match, MQRC_NO_MSG_AVAILABLE is returned. If MQGMO_BROWSE_NEXT is coded with MQMO_MATCH_MSG_TOKEN, the message identified by the *MsgToken* is returned only if it is beyond the browse-cursor for the calling handle.

If MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR is specified, MQMO_MATCH_MSG_TOKEN is ignored.

MQMO_MATCH_MSG_TOKEN is not valid with the following get message options:

- MQGMO_WAIT
- MQGMO_SET_SIGNAL

For an MQGET call specifying MQMO_MATCH_MSG_TOKEN, an MQGMO of version 3 or later must be supplied to the call, otherwise MQRC_WRONG_GMO_VERSION is returned.

If the *MsgToken* is not valid at this time, MQCC_FAILED with MQRC_NO_MSG_AVAILABLE is returned, unless there is another error.

Options (MQLONG)

MQGMO options control the action of MQGET. You can specify zero or more of the options. If you need more than one optional value:

- Add the values (do not add the same constant more than once), or
- Combine the values using the bitwise OR operation (if the programming language supports bit operations).

Combinations of options that are not valid are noted; all other combinations are valid.

Wait options: The following options relate to waiting for messages to arrive on the queue:

MQGMO_WAIT

The application waits until a suitable message arrives. The maximum time that the application waits is specified in *WaitInterval* .

Important: There is no wait, or delay, if a suitable message is available immediately.

If MQGET requests are inhibited, or MQGET requests become inhibited while waiting, the wait is canceled. The call completes with MQCC_FAILED and reason code MQRC_GET_INHIBITED, regardless of whether there are suitable messages on the queue.

You can use MQGMO_WAIT with the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT options.

If several applications are waiting on the same shared queue, the following rules select which application is activated when a suitable message arrives:

Table 494. Rules for activating MQGET calls on a shared queue.		
Number of MQGET calls waiting to be activated		Result
With a BROWSE option	Without a BROWSE option ¹	
None	One or more	One MQGET call without a BROWSE option is activated.
One or more	None	All MQGET calls with a BROWSE option are activated.
One or more	One or more	One MQGET call without a BROWSE option is activated. The number of MQGET calls with a BROWSE option that are activated is unpredictable.

If more than one MQGET call without a BROWSE option is waiting on the same queue, only one is activated. The queue manager attempts to give priority to waiting calls in the following order:

1. Specific get-wait requests that can be satisfied only by certain messages, for example, ones with a specific *MsgId* or *CorrelId* (or both).
2. General get-wait requests that can be satisfied by any message.

¹ An MQGET call specifying the MQGMO_LOCK option is treated as a nonbrowse call.

Note:

- Within the first category, no additional priority is given to more specific get-wait requests. For example, requests that specify both *MsgId* and *CorrelId* .
- Within either category, it cannot be predicted which application is selected. In particular, the application waiting longest is not necessarily the one selected.
- Path length, and priority-scheduling considerations of the operating system, can mean that a waiting application of lower operating system priority than expected retrieves the message.
- It can also happen that an application that is not waiting retrieves the message in preference to one that is.

On z/OS, the following points apply:

- If you want the application to proceed with other work while waiting for the message to arrive, consider using the signal option (`MQGMO_SET_SIGNAL`) instead. However the signal option is environment-specific; applications that you port between different environments must not use it.
- If there is more than one MQGET call waiting for the same message, with a mixture of wait and signal options, each waiting call is considered equally. It is an error to specify `MQGMO_SET_SIGNAL` with `MQGMO_WAIT` . It is also an error to specify this option with a queue handle for which a signal is outstanding.
- If you specify `MQGMO_WAIT` or `MQGMO_SET_SIGNAL` for a queue that has an *IndexType* of `MQIT_MSG_TOKEN`, no selection criteria are permitted. This means that:
 - If you are using a version-1 MQGMO, set the *MsgId* and *CorrelId* fields in the MQMD specified on the MQGET call to `MQMI_NONE` and `MQCI_NONE`.
 - If you are using a version-2 or later MQGMO, set the *MatchOptions* field to `MQMO_NONE`.
- For an MQGET call on a shared queue and the call is a browse request, or a destructive get of a group message, and neither *MsgId* nor *CorrelId* are to be matched, your signal ECB is posted `MQEC_MSG_ARRIVED` after 200 milliseconds.

This occurs, even though a suitable message might not have arrived on the queue, until the wait interval has expired, when the queue is posted with `MQEC_WAIT_INTERVAL_EXPIRED`. When `MQEC_MSG_ARRIVED` is posted, you must reissue a second MQGET call to retrieve the message, if one is available.

This technique is used to ensure that you are informed in a timely manner of a message arrival, but can appear as an unexpected processing overhead when compared with a similar call sequence on a nonshared queue.

`MQGMO_WAIT` is ignored if specified with `MQGMO_BROWSE_MSG_UNDER_CURSOR` or `MQGMO_MSG_UNDER_CURSOR` ; no error is raised.

MQGMO_NO_WAIT

The application does not wait if no suitable message is available. `MQGMO_NO_WAIT` is the opposite of the `MQGMO_WAIT`. `MQGMO_NO_WAIT` is defined to aid program documentation. It is the default if neither is specified.

MQGMO_SET_SIGNAL

Use this option with the *Signal1* and *Signal2* fields. It allows applications to proceed with other work while waiting for a message to arrive. It also allows (if suitable operating system facilities are available) applications to wait for messages arriving on more than one queue.

Note: The `MQGMO_SET_SIGNAL` option is environment-specific; do not use it for applications that you want to port.

In two circumstances, the call completes in the same way as if this option had not been specified:

1. If a currently available message satisfies the criteria specified in the message descriptor.
2. If a parameter error or other synchronous error is detected.

If no message satisfying the criteria specified in the message descriptor is currently available, control returns to the application without waiting for a message to arrive. The *CompCode* and *Reason*

parameters are set to `MQCC_WARNING` and `MQRC_SIGNAL_REQUEST_ACCEPTED`. Other output fields in the message descriptor and the output parameters of the `MQGET` call are not set. When a suitable message arrives later, the signal is delivered by posting the ECB.

The caller must then reissue the `MQGET` call to retrieve the message. The application can wait for this signal, using functions provided by the operating system.

If the operating system provides a multiple wait mechanism, you can use it to wait for a message arriving on any one of several queues.

If a nonzero *WaitInterval* is specified, the signal is delivered after the wait interval expires. The queue manager can also cancel the wait, in which case the signal is delivered.

More than one `MQGET` call can set a signal for the same message. The order in which applications are activated is the same as described for `MQGMO_WAIT`.

If more than one `MQGET` call is waiting for the same message, each waiting call is considered equally. The calls can include a mixture of wait and signal options.

Under certain conditions the `MQGET` call can retrieve a message, and a signal resulting from the arrival of the same message can be delivered. When a signal is delivered, an application must be prepared for no message to be available.

A queue handle can have no more than one signal request outstanding.

This option is not valid with any of the following options:

- `MQGMO_UNLOCK`
- `MQGMO_WAIT`

For an `MQGET` call on a shared queue and the call is a browse request, or a destructive get of a group message, and neither *MsgId* or *CorrelId* are to be matched, the user's signal ECB is posted `MQEC_MSG_ARRIVED` after 200 milliseconds.

This occurs, even though a suitable message might not have arrived on the queue, until the wait interval has expired, when the queue is posted with `MQEC_WAIT_INTERVAL_EXPIRED`. When `MQEC_MSG_ARRIVED` is posted, you must reissue a second `MQGET` call to retrieve the message, if one is available.

This technique is used to ensure that you are informed in a timely manner of a message arrival, but can appear as an unexpected processing overhead when compared with a similar call sequence on a nonshared queue.

This is not an efficient method of message retrieval when messages are added infrequently. To avoid this overhead for the browse case, specify *MsgId* (if non-indexed or indexed by *MsgId*) or *CorrelId* (if indexed by *CorrelId*) matching on the `MQGET` call.

This option is supported on z/OS only.

MQGMO_FAIL_IF QUIESCING

Force the `MQGET` call to fail if the queue manager is in the quiescing state.

On z/OS, this option also forces the `MQGET` call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

If this option is specified with `MQGMO_WAIT` or `MQGMO_SET_SIGNAL`, and the wait or signal is outstanding at the time the queue manager enters the quiescing state:

- The wait is canceled and the call returns completion code `MQCC_FAILED` with reason code `MQRC_Q_MGR QUIESCING` or `MQRC_CONNECTION QUIESCING`.
- The signal is canceled with an environment-specific signal completion code.

On z/OS, the signal completes with event completion code `MQEC_Q_MGR QUIESCING` or `MQEC_CONNECTION QUIESCING`.

If `MQGMO_FAIL_IF QUIESCING` is not specified and the queue manager or connection enters the quiescing state, the wait or signal is not canceled.

Sync point options: The following options relate to the participation of the MQGET call within a unit of work:

MQGMO_SYNCPOINT

The request is to operate within the normal unit-of-work protocols. The message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

You can leave MQGMO_SYNCPOINT and MQGMO_NO_SYNCPOINT unset. In which case, the inclusion of the get request in unit-of-work protocols is determined by the environment running the queue manager. It is not determined by the environment running the application. On z/OS, the get request is within a unit of work. In all other environments, the get request is not within a unit of work.

Because of these differences, an application that you want to port must not allow this option to default; specify MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT explicitly.

This option is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT
- MQGMO_LOCK
- MQGMO_NO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

MQGMO_SYNCPOINT_IF_PERSISTENT

The request is to operate within the normal unit-of-work protocols, but *only* if the message retrieved is persistent. A persistent message has the value MQPER_PERSISTENT in the *Persistence* field in MQMD.

- If the message is persistent, the queue manager processes the call as though the application had specified MQGMO_SYNCPOINT.
- If the message is not persistent, the queue manager processes the call as though the application had specified MQGMO_NO_SYNCPOINT.

This option is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT
- MQGMO_COMPLETE_MSG
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_NO_SYNCPOINT
- MQGMO_SYNCPOINT
- MQGMO_UNLOCK

This option is supported in the following environments: AIX, HP-UX, z/OS, IBM i, Solaris, and Linux, plus IBM MQ MQI clients connected to these systems.

MQGMO_NO_SYNCPOINT

The request is to operate outside the normal unit-of-work protocols. If you get a message without a browse option, it is deleted from the queue immediately. The message cannot be made available again by backing out the unit of work.

This option is assumed if you specify MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT.

You can leave MQGMO_SYNCPOINT and MQGMO_NO_SYNCPOINT unset. In which case, the inclusion of the get request in unit-of-work protocols is determined by the environment running the queue

manager. It is not determined by the environment running the application. On z/OS, the get request is within a unit of work. In all other environments, the get request is not within a unit of work.

Because of these differences, an application that you want to port must not allow this option to default; specify either MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT explicitly.

This option is not valid with any of the following options:

- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT

MQGMO_MARK_SKIP_BACKOUT

Back out a unit of work without reinstating on the queue the message that was marked with this option.

This option is supported only on z/OS.

If this option is specified, MQGMO_SYNCPOINT must also be specified. MQGMO_MARK_SKIP_BACKOUT is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT
- MQGMO_LOCK
- MQGMO_NO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

Note: On IMS and CICS, you might have to issue an extran IBM MQ call after backing out a unit of work containing a message marked with MQGMO_MARK_SKIP_BACKOUT. You must issue an IBM MQ call before you commit the new unit of work containing the marked message. The call can be any IBM MQ call you like.

1. On IMS, if you have not applied IMS APAR PN60855 and you are running an IMS MPP or BMP application.
2. On CICS, if you are running any application.

In both cases, issue any IBM MQ call before committing the new unit of work containing the backed out message.

Note: Within a unit of work, there can be only one get request marked as skipping backout, as well as none or several unmarked get requests.

If an application backs out of a unit of work, a message that was retrieved using MQGMO_MARK_SKIP_BACKOUT is not restored to its previous state. Other resource updates are backed out. The message is treated as if it had been retrieved in a new unit of work started by the backout request. The message is retrieved without the MQGMO_MARK_SKIP_BACKOUT option.

MQGMO_MARK_SKIP_BACKOUT is useful if, after some resources have been changed, it becomes apparent that the unit of work cannot complete successfully. If you omit this option, backing out the unit of work reinstates the message on the queue. The same sequence of events occurs again, when the message is next retrieved.

However, if you specify MQGMO_MARK_SKIP_BACKOUT on the original MQGET call, backing out the unit of work backs out the updates to the other resources. The message is treated as if it had been retrieved under a new unit of work. The application can perform appropriate error handling. It can send a report message to the sender of the original message, or place the original message on the dead-letter queue. It can then commit the new unit of work. Committing the new unit of work removes the message permanently from the original queue.

MQGMO_MARK_SKIP_BACKOUT marks a single physical message. If the message belongs to a message group, the other messages in the group are not marked. Similarly, if the marked message is a segment of a logical message, the other segments in the logical message are not marked.

Any message in a group can be marked, but if messages are retrieved using MQGMO_LOGICAL_ORDER, it is advantageous to mark the first message in the group. If the unit of work is backed out, the first (marked) message is moved to the new unit of work. The second and later messages in the group are reinstated on the queue. The messages left on the queue cannot be retrieved by another application using MQGMO_LOGICAL_ORDER. The first message in the group is no longer on the queue. However, the application that backed the unit of work out can retrieve the second and later messages into the new unit of work using the MQGMO_LOGICAL_ORDER option. The first message has been retrieved already.

Occasionally you might need to back out the new unit of work. For example, because the dead-letter queue is full and the message must not be discarded. Backing out the new unit of work reinstates the message on the original queue, which prevents the message being lost. However, in this situation processing cannot continue. After backing out the new unit of work, the application must inform the operator or administrator that there is an unrecoverable error, and then finish.

MQGMO_MARK_SKIP_BACKOUT only works if the unit of work containing the get request is interrupted by the application backing it out. If the unit of work containing the get request is backed out because the transaction or system fails, MQGMO_MARK_SKIP_BACKOUT is ignored. Any message retrieved using this option is reinstated on the queue in the same way as messages retrieved without this option.

Browse options: The following options relate to browsing messages on the queue:

MQGMO_BROWSE_FIRST

When a queue is opened with the MQOO_BROWSE option, a browse cursor is established, positioned logically before the first message on the queue. You can then use MQGET calls specifying the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT, or MQGMO_BROWSE_MSG_UNDER_CURSOR option to retrieve messages from the queue nondestructively. The browse cursor marks the position, within the messages on the queue, from which the next MQGET call with MQGMO_BROWSE_NEXT searches for a suitable message.

MQGMO_BROWSE_FIRST is not valid with any of the following options:

- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_MSG_UNDER_CURSOR
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

An MQGET call with MQGMO_BROWSE_FIRST ignores the previous position of the browse cursor. The first message on the queue that satisfies the conditions specified in the message descriptor is retrieved. The message remains on the queue, and the browse cursor is positioned on this message.

After this call, the browse cursor is positioned on the message that has been returned. The message might be removed from the queue before the next MQGET call with MQGMO_BROWSE_NEXT is issued. In this case, the browse cursor remains at the position in the queue that the message occupied, even though that position is now empty.

Use the MQGMO_MSG_UNDER_CURSOR option with a non-browse MQGET call, to remove the message from the queue.

The browse cursor is not moved by a non-browse MQGET call, even if using the same *Hobj* handle. Nor is it moved by a browse MQGET call that returns a completion code of MQCC_FAILED, or a reason code of MQRC_TRUNCATED_MSG_FAILED.

Specify the MQGMO_LOCK option with this option, to lock the message that is browsed.

You can specify MQGMO_BROWSE_FIRST with any valid combination of the MQGMO_* and MQMO_* options that control the processing of messages in groups and segments of logical messages.

If you specify MQGMO_LOGICAL_ORDER, the messages are browsed in logical order. If you omit that option, the messages are browsed in physical order. If you specify MQGMO_BROWSE_FIRST, you can switch between logical order and physical order. Subsequent MQGET calls using MQGMO_BROWSE_NEXT browse the queue in the same order as the most recent call that specified MQGMO_BROWSE_FIRST for the queue handle.

The queue manager retains two sets of group and segment information for MQGET calls. The group and segment information for browse calls are retained separately from the information for calls that remove messages from the queue. If you specify MQGMO_BROWSE_FIRST, the queue manager ignores the group and segment information for browsing. It scans the queue as though there were no current group and no current logical message. If the MQGET call is successful, completion code MQCC_OK or MQCC_WARNING, the group and segment information for browsing is set to that of the message returned. If the call fails, the group and segment information remain the same as they were before the call.

MQGMO_BROWSE_NEXT

Advance the browse cursor to the next message on the queue that satisfies the selection criteria specified on the MQGET call. The message is returned to the application, but remains on the queue.

MQGMO_BROWSE_NEXT is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_MSG_UNDER_CURSOR
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

MQGMO_BROWSE_NEXT behaves the same way as MQGMO_BROWSE_FIRST, if it is the first call to browse a queue, after the queue has been opened for browse.

The message under cursor might be removed from the queue before the next MQGET call with MQGMO_BROWSE_NEXT is issued. The browse cursor logically remains at the position in the queue that the message occupied, even though that position is now empty.

Messages are stored on the queue in one of two ways:

- FIFO within priority (MQMDS_PRIORITY), or
- FIFO *regardless* of priority (MQMDS_FIFO)

The *MsgDeliverySequence* queue attribute indicates which method applies (see [“Attributes for queues”](#) on page 801 for details).

A queue might have a *MsgDeliverySequence* of MQMDS_PRIORITY. A message arrives on the queue that is of a higher priority than the one currently pointed to by the browse cursor. In which case, the higher priority message is not found during the current sweep of the queue using MQGMO_BROWSE_NEXT. It can be found only after the browse cursor has been reset with MQGMO_BROWSE_FIRST, or by reopening the queue.

The MQGMO_MSG_UNDER_CURSOR option can be used with a non-browse MQGET call if required, to remove the message from the queue.

The browse cursor is not moved by non-browse MQGET calls using the same *Hobj* handle.

Specify the MQGMO_LOCK option with this option to lock the message that is browsed.

You can specify MQGMO_BROWSE_NEXT with any valid combination of the MQGMO_* and MQMO_* options that control the processing of messages in groups and segments of logical messages.

If you specify MQGMO_LOGICAL_ORDER, the messages are browsed in logical order. If you omit that option, the messages are browsed in physical order. If you specify MQGMO_BROWSE_FIRST, you can switch between logical order and physical order. Subsequent MQGET calls using MQGMO_BROWSE_NEXT browse the queue in the same order as the most recent call that specified MQGMO_BROWSE_FIRST for the queue handle. The call fails with reason code MQRC_INCONSISTENT_BROWSE if this condition is not satisfied.

Note: Take special care when using an MQGET call to browse beyond the end of a message group if MQGMO_LOGICAL_ORDER is not specified. For example, suppose that the last message in the group precedes the first message in the group on the queue. Using MQGMO_BROWSE_NEXT to browse beyond the end of the group, specifying MQMO_MATCH_MSG_SEQ_NUMBER with *MsgSeqNumber* set to 1 returns the first message in the group already browsed. This result can happen immediately, or a number of MQGET calls later if there are intervening groups. The same consideration applies for a logical message not in a group.

The group and segment information for browse calls are retained separately from the information for calls that remove messages from the queue.

MQGMO_BROWSE_MSG_UNDER_CURSOR

Retrieve the message pointed to by the browse cursor nondestructively, regardless of the MQMO_* options specified in the *MatchOptions* field in MQGMO.

MQGMO_BROWSE_MSG_UNDER_CURSOR is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_NEXT
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_MSG_UNDER_CURSOR
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

The message pointed to by the browse cursor is the one that was last retrieved using either the MQGMO_BROWSE_FIRST or the MQGMO_BROWSE_NEXT option. The call fails if neither of these calls has been issued for this queue since it was opened. The call also fails if the message that was under the browse cursor has since been retrieved destructively.

The position of the browse cursor is not changed by this call.

The MQGMO_MSG_UNDER_CURSOR option can be used with a non-browse MQGET call, to remove the message from the queue.

The browse cursor is not moved by a non-browse MQGET call, even if using the same *Hobj* handle. Nor is it moved by a browse MQGET call that returns a completion code of MQCC_FAILED, or a reason code of MQRC_TRUNCATED_MSG_FAILED.

If MQGMO_BROWSE_MSG_UNDER_CURSOR is specified with MQGMO_LOCK:

- If there is already a message locked, it must be the one under the cursor, so that is returned without unlocking and locking again. The message remains locked.
- If there is no locked message and there is a message under the browse cursor, it is locked and returned to the application. If there is no message under the browse cursor, the call fails.

If MQGMO_BROWSE_MSG_UNDER_CURSOR is specified without MQGMO_LOCK:

- If there is already a message locked, it must be the one under the cursor. The message is returned to the application and then unlocked. Because the message is now unlocked, there is no guarantee

that it can be browsed again, or retrieved destructively by the same application. It might have been retrieved destructively by another application getting messages from the queue.

- If there is no locked message and there is a message under the browse cursor, it is returned to the application. If there is no message under the browse cursor the call fails.

If MQGMO_COMPLETE_MSG is specified with MQGMO_BROWSE_MSG_UNDER_CURSOR, the browse cursor must identify a message whose *Offset* field in MQMD is zero. If this condition is not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

The group and segment information for browse calls are retained separately from the information for calls that remove messages from the queue.

MQGMO_MSG_UNDER_CURSOR

Retrieve the message pointed to by the browse cursor, regardless of the MQMO_* options specified in the *MatchOptions* field in MQGMO. The message is removed from the queue.

The message pointed to by the browse cursor is the one that was last retrieved using either the MQGMO_BROWSE_FIRST or the MQGMO_BROWSE_NEXT option.

If MQGMO_COMPLETE_MSG is specified with MQGMO_MSG_UNDER_CURSOR, the browse cursor must identify a message whose *Offset* field in MQMD is zero. If this condition is not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

This option is not valid with any of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT
- MQGMO_UNLOCK

It is also an error if the queue was not opened both for browse and for input. If the browse cursor is not currently pointing to a retrievable message, an error is returned by the MQGET call.

MQGMO_MARK_BROWSE_HANDLE

The message that is returned by a successful MQGET, or identified by the returned *MsgToken*, is marked. The mark is specific to the object handle used in the call.

The message is not removed from the queue.

MQGMO_MARK_BROWSE_HANDLE is valid only if one of the following options is also specified:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT

MQGMO_MARK_BROWSE_HANDLE is not valid with any of the following options:

- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_COMPLETE_MSG
- MQGMO_LOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_UNLOCK

The message remains in this state until one of the following events occurs:

- The object handle concerned is closed, either normally or otherwise.
- The message is unmarked for this handle by a call to MQGET with the option MQGMO_UNMARK_BROWSE_HANDLE.
- The message is returned from a call to destructive MQGET, which completes with MQCC_OK or MQCC_WARNING. The message state remains changed even if the MQGET is later rolled-back.

- The message expires.

MQGMO_MARK_BROWSE_CO_OP

The message that is returned by a successful MQGET, or identified by the returned *MsgToken*, is marked for all handles in the cooperating set.

The cooperative level mark is in addition to any handle level mark that might have been set.

The message is not removed from the queue.

MQGMO_MARK_BROWSE_CO_OP is valid only if the object handle used was returned by a call to MQOPEN that specified MQOO_CO_OP. You must also specify one of the following MQGMO options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_BROWSE_NEXT

This option is not valid with any of the following options:

- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_COMPLETE_MSG
- MQGMO_LOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_UNLOCK

If the message is already marked, and the option MQGMO_UNMARKED_BROWSE_MSG is not specified, the call fails with MQCC_FAILED and reason code MQRC_MSG_MARKED_BROWSE_CO_OP.

The message remains in this state until one of the following events occurs:

- All object handles in the cooperating set are closed.
- The message is unmarked for cooperating browsers by a call to MQGET with the option MQGMO_UNMARK_BROWSE_CO_OP.
- The message is automatically unmarked by the queue manager.
- The message is returned from a call to a non-browse MQGET. The message state remains changed even if the MQGET is later rolled-back.
- The message expires.

MQGMO_UNMARKED_BROWSE_MSG

A call to MQGET that specifies MQGMO_UNMARKED_BROWSE_MSG returns a message that is considered to be unmarked for its handle. It does not return a message if the message was marked for its handle. It also does not return the message if the queue was opened by a call to MQOPEN, with the option MQOO_CO_OP, and the message has been marked by a member of the cooperating set.

This option is not valid with any of the following options:

- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_COMPLETE_MSG
- MQGMO_LOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_UNLOCK

MQGMO_UNMARK_BROWSE_CO_OP

After a call to MQGET that specifies this option, the message is no longer considered by any open handles in the set of cooperating handles to be marked for the cooperating set. The message is still considered to be marked at handle level if it was marked at handle level before this call.

Using MQGMO_UNMARK_BROWSE_CO_OP is valid only with a handle returned by a successful call to MQOPEN with the option MQOO_CO_OP. The MQGET succeeds even if the message is not considered to be marked by the cooperating set of handles.

MQGMO_UNMARK_BROWSE_CO_OP is not valid on a non-browse MQGET call, or with any of the following options:

- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_COMPLETE_MSG
- MQGMO_LOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_MARK_BROWSE_CO_OP
- MQGMO_UNLOCK
- MQGMO_UNMARKED_BROWSE_MSG

MQGMO_UNMARK_BROWSE_HANDLE

After a call to MQGET that specifies this option, the message located is no longer considered to be marked by this handle.

The call succeeds even if the message is not marked for this handle.

This option is not valid on a non-browse MQGET call, or with any of the following options:

- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_COMPLETE_MSG
- MQGMO_LOCK
- MQGMO_LOGICAL_ORDER
- MQGMO_MARK_BROWSE_CO_OP
- MQGMO_UNLOCK
- MQGMO_UNMARKED_BROWSE_MSG

Lock options: The following options relate to locking messages on the queue:

MQGMO_LOCK

Lock the message that is browsed, so that the message becomes invisible to any other handle open for the queue. The option can be specified only if one of the following options is also specified:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_NEXT
- MQGMO_BROWSE_MSG_UNDER_CURSOR

Only one message can be locked for each queue handle. The message can be a logical message or a physical message:

- If you specify MQGMO_COMPLETE_MSG, all the message segments that make up the logical message are locked to the queue handle. The messages must all be present on the queue and available for retrieval.
- If you omit MQGMO_COMPLETE_MSG, only a single physical message is locked to the queue handle. If this message happens to be a segment of a logical message, the locked segment prevents other applications using MQGMO_COMPLETE_MSG to retrieve or browse the logical message.

The locked message is always the one under the browse cursor. The message can be removed from the queue by a later MQGET call that specifies the MQGMO_MSG_UNDER_CURSOR option. Other MQGET calls using the queue handle can also remove the message (for example, a call that specifies the message identifier of the locked message).

If the call returns completion code MQCC_FAILED, or MQCC_WARNING with reason code MQRC_TRUNCATED_MSG_FAILED, no message is locked.

If the application does not remove the message from the queue, the lock is released by one of the following actions:

- Issuing another MQGET call for this handle, specifying either MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT. The lock is released if the call completes with MQCC_OK or MQCC_WARNING. The message remains locked if the call completes with MQCC_FAILED. However, the following exceptions apply:
 - The message is not unlocked if MQCC_WARNING is returned with MQRC_TRUNCATED_MSG_FAILED.
 - The message is unlocked if MQCC_FAILED is returned with MQRC_NO_MSG_AVAILABLE.

If you also specify MQGMO_LOCK, the message returned is locked. If you omit MQGMO_LOCK, there is no locked message after the call.

If you specify MQGMO_WAIT, and no message is immediately available, the original message is unlocked before the start of the wait.

- Issuing another MQGET call for this handle, with MQGMO_BROWSE_MSG_UNDER_CURSOR, without MQGMO_LOCK. The lock is released if the call completes with MQCC_OK or MQCC_WARNING. The message remains locked if the call completes with MQCC_FAILED. However, the following exception applies:
 - The message is not unlocked if MQCC_WARNING is returned with MQRC_TRUNCATED_MSG_FAILED.
- Issuing another MQGET call for this handle with MQGMO_UNLOCK.
- Issuing an MQCLOSE call using the handle. The MQCLOSE might be implicit, caused by the application ending.

No special MQOPEN option is required to specify MQGMO_LOCK, other than MQ00_BROWSE, which is needed to specify an accompanying browse option.

MQGMO_LOCK is not valid with any of the following options:

- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_UNLOCK

MQGMO_LOCK is not possible when you are using an IBM MQ client on HP Integrity NonStop Server to a z/OS queue manager when coordinated by TMF.

MQGMO_UNLOCK

The message to be unlocked must have been previously locked by an MQGET call with the MQGMO_LOCK option. If there is no message locked for this handle, the call completes with MQCC_WARNING and MQRC_NO_MSG_LOCKED.

The *MsgDesc*, *BufferLength*, *Buffer*, and *DataLength* parameters are not checked or altered if you specify MQGMO_UNLOCK. No message is returned in *Buffer*.

No special open option is required to specify MQGMO_UNLOCK (although MQ00_BROWSE is needed to issue the lock request in the first place).

This option is not valid with any options except the following:

- MQGMO_NO_WAIT
- MQGMO_NO_SYNCPOINT

Both of these options are assumed whether specified or not.

Message-data options: The following options relate to the processing of the message data when the message is read from the queue:

MQGMO_ACCEPT_TRUNCATED_MSG

If the message buffer is too small to hold the complete message, allow the MQGET call to fill the buffer. MQGET fills the buffer with as much of the message it can. It issues a warning completion code, and completes its processing. This means that:

- When browsing messages, the browse cursor is advanced to the returned message.
- When removing messages, the returned message is removed from the queue.
- Reason code MQRC_TRUNCATED_MSG_ACCEPTED is returned if no other error occurs.

Without this option, the buffer is still filled with as much of the message as it can hold. A warning completion code is issued, but processing is not completed. This means that:

- When browsing messages, the browse cursor is not advanced.
- When removing messages, the message is not removed from the queue.
- Reason code MQRC_TRUNCATED_MSG_FAILED is returned if no other error occurs.

MQGMO_CONVERT

This option converts the application data in the message to conform to the *CodedCharSetId* and *Encoding* values specified in the *MsgDesc* parameter on the MQGET call. The data is converted before it is copied to the *Buffer* parameter.

The *Format* field specified when the message was put is assumed by the conversion process to identify the nature of the data in the message. The message data is converted by the queue manager for built-in formats, and by a user-written exit for other formats. See [“Data-conversion exit” on page 870](#) for details of the data-conversion exit.

- If conversion is successful, the *CodedCharSetId* and *Encoding* fields specified in the *MsgDesc* parameter are unchanged on return from the MQGET call.
- If only conversion fails the message data is returned unconverted. The *CodedCharSetId* and *Encoding* fields in *MsgDesc* are set to the values for the unconverted message. The completion code is MQCC_WARNING in this case.

In either case, these fields describe the character-set identifier and encoding of the message data that is returned in the *Buffer* parameter.

See the *Format* field described in [“MQMD - Message descriptor” on page 396](#) for a list of format names for which the queue manager performs the conversion.

Group and segment options: The following options relate to the processing of messages in groups and segments of logical messages. Before the option descriptions, here are some definitions of important terms:

Physical message

A physical message is the smallest unit of information that can be placed on or removed from a queue. It often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor, MQMD. Typically, physical messages are distinguished by differing values for the message identifier, the *MsgId* field in MQMD. The queue manager does not enforce different values.

Logical message

A logical message is a single unit of application information. In the absence of system constraints, a logical message is the same as a physical message. If logical messages are large, system constraints might make it advisable or necessary to split a logical message into two or more physical messages, called segments.

A logical message that has been segmented consists of two or more physical messages that have the same nonnull group identifier, *GroupId* field in MQMD. They have the same message sequence number, *MsgSeqNumber* field in MQMD. The segments are distinguished by differing values for the segment offset, *Offset* field in MQMD. The segment offset is the offset of the data in the physical

message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message typically have different message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a nonnull group identifier. In this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages, for which segmentation has been inhibited by the sending application, have a null group identifier, MQGI_NONE, unless the logical message belongs to a message group.

Message group

A message group is a set of one or more logical messages that have the same nonnull group identifier. The logical messages in the group are distinguished by different values for the message sequence number. The sequence number is an integer in the range 1 through n, where n is the number of logical messages in the group. If one or more of the logical messages is segmented, there are more than n physical messages in the group.

MQGMO_LOGICAL_ORDER

MQGMO_LOGICAL_ORDER controls the order in which messages are returned by successive MQGET calls for the queue handle. The option must be specified on each call.

If MQGMO_LOGICAL_ORDER is specified for successive MQGET calls for the same queue handle, messages in groups are returned in the order of their message sequence numbers. Segments of logical messages are returned in the order given by their segment offsets. This order might be different from the order in which those messages and segments occur on the queue.

Note: Specifying MQGMO_LOGICAL_ORDER has no adverse consequences on messages that do not belong to groups and that are not segments. In effect, such messages are treated as though each belonged to a message group consisting of only one message. It is safe to specify MQGMO_LOGICAL_ORDER when retrieving messages from queues that contain a mixture of messages in groups, message segments, and unsegmented messages not in groups.

To return the messages in the required order, the queue manager retains the group and segment information between successive MQGET calls. The group and segment information identifies the current message group and current logical message for the queue handle. It also identifies the current position within the group and logical message, and whether the messages are being retrieved within a unit of work. Because the queue manager retains this information, the application does not need to set the group and segment information before each MQGET call. Specifically, it means that the application does not need to set the *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD. However, the application must set the MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT option correctly on each call.

When the queue is opened, there is no current message group and no current logical message. A message group becomes the current message group when a message that has the MQMF_MSG_IN_GROUP flag is returned by the MQGET call. With MQGMO_LOGICAL_ORDER specified on successive calls, that group remains the current group until a message is returned that has:

- MQMF_LAST_MSG_IN_GROUP without MQMF_SEGMENT (that is, the last logical message in the group is not segmented), or
- MQMF_LAST_MSG_IN_GROUP with MQMF_LAST_SEGMENT (that is, the message returned is the last segment of the last logical message in the group).

When such a message is returned, the message group is terminated, and on successful completion of the MQGET call there is no longer a current group. In a similar way, a logical message becomes the current logical message when a message that has the MQMF_SEGMENT flag is returned by the MQGET call. The logical message is terminated when the message that has the MQMF_LAST_SEGMENT flag is returned.

If no selection criteria are specified, successive MQGET calls return, in the correct order, the messages for the first message group on the queue. They then return the messages for the second message group, and so on, until there are no more messages available. It is possible to select the particular message groups returned by specifying one or more of the following options in the *MatchOptions* field:

- MQMO_MATCH_MSG_ID
- MQMO_MATCH_CORREL_ID
- MQMO_MATCH_GROUP_ID

However, these options are effective only when there is no current message group or logical message. See the *MatchOptions* field described in “MQGMO - Get-message options” on page 350 for further details.

Table 495 on page 369 shows the values of the *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* fields that the queue manager looks for when attempting to find a message to return on the MQGET call. The rules apply both to removing messages from the queue, and browsing messages on the queue. In the table, Either means Yes or No:

LOG ORD

Indicates whether the MQGMO_LOGICAL_ORDER option is specified on the call.

Cur grp

Indicates whether a current message group exists before the call.

Cur log msg

Indicates whether a current logical message exists before the call.

Other columns

Show the values that the queue manager looks for. Previous denotes the value returned for the field in the previous message for the queue handle.

Table 495. MQGET options relating to messages in groups and segments of logical messages							
Options you specify	Group and log-msg status before call		Values the queue manager looks for				
LOG ORD	Cur grp	Cur log msg	<i>MsgId</i>	<i>CorrelId</i>	<i>GroupId</i>	<i>MsgSeqNumber</i>	<i>Offset</i>
Yes	No	No	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>	1	0
Yes	No	Yes	Any message identifier	Any correlation identifier	Previous group identifier	1	Previous offset + previous segment length
Yes	Yes	No	Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number + 1	0
Yes	Yes	Yes	Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number	Previous offset + previous segment length
No	Either	Either	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>	Controlled by <i>MatchOptions</i>

If multiple message groups are present on the queue and eligible for return, the groups are returned in the order determined by the position on the queue of the first segment of the first logical message in each group. That is, the physical messages that have message sequence numbers of 1, and offsets of 0, determine the order in which eligible groups are returned.

The MQGMO_LOGICAL_ORDER option affects units of work as follows:

- If the first logical message or segment in a group is retrieved within a unit of work, all the other logical messages and segments in the group must be retrieved within a unit of work, if the same queue handle is used. However, they need not be retrieved within the same unit of work. This allows a message group consisting of many physical messages to be split across two or more consecutive units of work for the queue handle.
- If the first logical message or segment in a group is *not* retrieved within a unit of work, and the same queue handle is used, none of the other logical messages and segments in the group can be retrieved within a unit of work.

If these conditions are not satisfied, the MQGET call fails with reason code MQRC_INCONSISTENT_UOW.

When MQGMO_LOGICAL_ORDER is specified, the MQGMO supplied on the MQGET call must not be less than MQGMO_VERSION_2, and the MQMD must not be less than MQMD_VERSION_2. If this condition is not satisfied, the call fails with reason code MQRC_WRONG_GMO_VERSION or MQRC_WRONG_MD_VERSION, as appropriate.

If MQGMO_LOGICAL_ORDER is *not* specified for successive MQGET calls for the queue handle, messages are returned without regard for whether they belong to message groups, or whether they are segments of logical messages. This means that messages or segments from a particular group or logical message might be returned out of order, or intermingled with messages or segments from other groups or logical messages, or with messages that are not in groups and are not segments. In this situation, the particular messages that are returned by successive MQGET calls is controlled by the MQMO_* options specified on those calls (see the *MatchOptions* field described in [“MQGMO - Get-message options”](#) on page 350 for details of these options).

This is the technique that can be used to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the *GroupId*, *MsgSeqNumber*, *Offset*, and *MatchOptions* fields to the appropriate values, and then issue the MQGET call with MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT set, but *without* specifying MQGMO_LOGICAL_ORDER. If this call is successful, the queue manager retains the group and segment information, and subsequent MQGET calls using that queue handle can specify MQGMO_LOGICAL_ORDER as normal.

The group and segment information that the queue manager retains for the MQGET call is separate from the group and segment information that it retains for the MQPUT call. In addition, the queue manager retains separate information for:

- MQGET calls that remove messages from the queue.
- MQGET calls that browse messages on the queue.

For any given queue handle, the application can mix MQGET calls that specify MQGMO_LOGICAL_ORDER with MQGET calls that do not. However, note the following points:

- If you omit MQGMO_LOGICAL_ORDER, each successful MQGET call causes the queue manager to set the saved group and segment information to the values corresponding to the message returned; this replaces the existing group and segment information retained by the queue manager for the queue handle. Only the information appropriate to the action of the call (browse or remove) is modified.
- If you omit MQGMO_LOGICAL_ORDER, the call does not fail if there is a current message group or logical message; the call might succeed with an MQCC_WARNING completion code. [Table 496 on page 371](#) shows the various cases that can arise. In these cases, if the completion code is not MQCC_OK, the reason code is one of the following (as appropriate):
 - MQRC_INCOMPLETE_GROUP
 - MQRC_INCOMPLETE_MSG
 - MQRC_INCONSISTENT_UOW

Note: The queue manager does not check the group and segment information when browsing a queue, or when closing a queue that was opened for browse but not input; in those cases the completion code is always MQCC_OK (assuming no other errors).

Table 496. Outcome when MQGET or MQCLOSE call is not consistent with group and segment information		
Current call is	Previous call was MQGET with MQGMO_LOGICAL_ORDER	Previous call was MQGET without MQGMO_LOGICAL_ORDER
MQGET with MQGMO_LOGICAL_ORDER	MQCC_FAILED	MQCC_FAILED
MQGET without MQGMO_LOGICAL_ORDER	MQCC_WARNING	MQCC_OK
MQCLOSE with an unterminated group or logical message	MQCC_WARNING	MQCC_OK

Applications that want to retrieve messages and segments in logical order are recommended to specify MQGMO_LOGICAL_ORDER, as this is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications might need more control than that provided by the MQGMO_LOGICAL_ORDER option, and this can be achieved by not specifying that option. The application must then ensure that the *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD, and the MQMO_* options in *MatchOptions* in MQGMO, are set correctly, before each MQGET call.

For example, an application that wants to *forward* physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, must *not* specify MQGMO_LOGICAL_ORDER. In a complex network with multiple paths between sending and receiving queue managers, the physical messages might arrive out of order. By specifying neither MQGMO_LOGICAL_ORDER, nor the corresponding MQPMO_LOGICAL_ORDER on the MQPUT call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without having to wait for the next one in logical order to arrive.

You can specify MQGMO_LOGICAL_ORDER with any of the other MQGMO_* options, and with various of the MQMO_* options in appropriate circumstances (see preceding section).

- On z/OS, this option is supported for private and shared queues, but the queue must have an index type of MQIT_GROUP_ID. For shared queues, the CFSTRUCT object that the queue maps to must be at CFLEVEL(3) or higher.
- On AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems, this option is supported for all local queues.

MQGMO_COMPLETE_MSG

Only a complete logical message can be returned by the MQGET call. If the logical message is segmented, the queue manager reassembles the segments and returns the complete logical message to the application; the fact that the logical message was segmented is not apparent to the application retrieving it.

Note: This is the only option that causes the queue manager to reassemble message segments. If not specified, segments are returned individually to the application if they are present on the queue (and they satisfy the other selection criteria specified on the MQGET call). Applications that do not want to receive individual segments must always specify MQGMO_COMPLETE_MSG.

To use this option, the application must provide a buffer that is big enough to accommodate the complete message, or specify the MQGMO_ACCEPT_TRUNCATED_MSG option.

If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_COMPLETE_MSG prevents the retrieval of segments belonging to incomplete logical messages. However, those message segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there might be no retrievable logical messages, even though *CurrentQDepth* is greater than zero.

For *persistent* messages, the queue manager can reassemble the segments only within a unit of work:

- If the MQGET call is operating within a user-defined unit of work, that unit of work is used. If the call fails during the reassembly process, the queue manager reinstates on the queue any segments that were removed during reassembly. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically (the application does not need to do this). If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work exists, the queue manager cannot reassemble. If the message does not require reassembly, the call can still succeed. But if the message requires reassembly, the call fails with reason code MQRC_UOW_NOT_AVAILABLE.

For *nonpersistent* messages, the queue manager does not require a unit of work to be available to perform reassembly.

Each physical message that is a segment has its own message descriptor. For the segments constituting a single logical message, most of the fields in the message descriptor are the same for all segments in the logical message; typically it is only the *MsgId*, *Offset*, and *MsgFlags* fields that differ between segments in the logical message. However, if a segment is placed on a dead-letter queue at an intermediate queue manager, the DLQ handler retrieves the message specifying the MQGMO_CONVERT option, and this can result in the character set or encoding of the segment being changed. If the DLQ handler successfully sends the segment on its way, the segment might have a character set or encoding that differs from the other segments in the logical message when the segment arrives at the destination queue manager.

A logical message consisting of segments in which the *CodedCharSetId* and *Encoding* fields differ cannot be reassembled by the queue manager into a single logical message. Instead, the queue manager reassembles and returns the first few consecutive segments at the start of the logical message that have the same character-set identifiers and encodings, and the MQGET call completes with completion code MQCC_WARNING and reason code MQRC_INCONSISTENT_CCIDS or MQRC_INCONSISTENT_ENCODINGS, as appropriate. This happens regardless of whether MQGMO_CONVERT is specified. To retrieve the remaining segments, the application must reissue the MQGET call without the MQGMO_COMPLETE_MSG option, retrieving the segments one by one. MQGMO_LOGICAL_ORDER can be used to retrieve the remaining segments in order.

An application that puts segments can also set other fields in the message descriptor to values that differ between segments. However, there is no advantage in doing this if the receiving application uses MQGMO_COMPLETE_MSG to retrieve the logical message. When the queue manager reassembles a logical message, it returns in the message descriptor the values from the message descriptor for the *first* segment; the only exception is the *MsgFlags* field, which the queue manager sets to indicate that the reassembled message is the only segment.

If MQGMO_COMPLETE_MSG is specified for a report message, the queue manager performs special processing. The queue manager checks the queue to see if all the report messages of that report type relating to the different segments in the logical message are present on the queue. If they are, they can be retrieved as a single message by specifying MQGMO_COMPLETE_MSG. For this to be possible, either the report messages must be generated by a queue manager or MCA which supports segmentation, or the originating application must request at least 100 bytes of message data (that is, the appropriate MQRO_*_WITH_DATA or MQRO_*_WITH_FULL_DATA options must be specified). If less than the full amount of application data is present for a segment, the missing bytes are replaced by nulls in the report message returned.

If MQGMO_COMPLETE_MSG is specified with MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR, the browse cursor must be positioned on a message whose *Offset* field in MQMD has a value of 0. If this condition is not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

MQGMO_COMPLETE_MSG implies MQGMO_ALL_SEGMENTS_AVAILABLE, which need not therefore be specified.

MQGMO_COMPLETE_MSG can be specified with any of the other MQGMO_* options apart from MQGMO_SYNCPOINT_IF_PERSISTENT, and with any of the MQMO_* options apart from MQMO_MATCH_OFFSET.

- On z/OS, this option is supported for private and shared queues, but the queue must have an index type of MQIT_GROUP_ID. For shared queues, the CFSTRUCT object that the queue map to must be at CFLEVEL(3) or higher.
- On AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems, this option is supported for all local queues.

MQGMO_ALL_MSGS_AVAILABLE

Messages in a group become available for retrieval only when *all* messages in the group are available. If the queue contains message groups with some of the messages missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_ALL_MSGS_AVAILABLE prevents retrieval of messages belonging to incomplete groups. However, those messages still contribute to the value of the *CurrentQDepth* queue attribute; this means that there may be no retrievable message groups, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code MQRC_NO_MSG_AVAILABLE is returned after the specified wait interval (if any) has expired.

The processing of MQGMO_ALL_MSGS_AVAILABLE depends on whether MQGMO_LOGICAL_ORDER is also specified:

- If both options are specified, MQGMO_ALL_MSGS_AVAILABLE has an effect *only* when there is no current group or logical message. If there *is* a current group or logical message, MQGMO_ALL_MSGS_AVAILABLE is ignored. This means that MQGMO_ALL_MSGS_AVAILABLE can remain on when processing messages in logical order.
- If MQGMO_ALL_MSGS_AVAILABLE is specified without MQGMO_LOGICAL_ORDER, MQGMO_ALL_MSGS_AVAILABLE *always* has an effect. This means that the option must be turned off after the first message in the group has been removed from the queue, in order to be able to remove the remaining messages in the group.

Successful completion of an MQGET call specifying MQGMO_ALL_MSGS_AVAILABLE means that at the time that the MQGET call was issued, all the messages in the group were on the queue. However, be aware that other applications can still remove messages from the group (the group is not locked to the application that retrieves the first message in the group).

If you omit this option, messages belonging to groups can be retrieved even when the group is incomplete.

MQGMO_ALL_MSGS_AVAILABLE implies MQGMO_ALL_SEGMENTS_AVAILABLE, which need not therefore be specified.

MQGMO_ALL_MSGS_AVAILABLE can be specified with any of the other MQGMO_* options, and with any of the MQMO_* options.

- On z/OS, this option is supported for private and shared queues, but the queue must have an index type of MQIT_GROUP_ID. For shared queues, the CFSTRUCT object that the queue map to must be at CFLEVEL(3) or higher.
- On AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems, this option is supported for all local queues.

MQGMO_ALL_SEGMENTS_AVAILABLE

Segments in a logical message become available for retrieval only when *all* segments in the logical message are available. If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_ALL_SEGMENTS_AVAILABLE prevents retrieval of segments belonging to incomplete logical messages. However, those segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there might be no retrievable logical messages, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code MQRC_NO_MSG_AVAILABLE is returned after the specified wait interval (if any) has expired.

The processing of MQGMO_ALL_SEGMENTS_AVAILABLE depends on whether MQGMO_LOGICAL_ORDER is also specified:

- If both options are specified, MQGMO_ALL_SEGMENTS_AVAILABLE has an effect *only* when there is no current logical message. If there *is* a current logical message, MQGMO_ALL_SEGMENTS_AVAILABLE is ignored. This means that MQGMO_ALL_SEGMENTS_AVAILABLE can remain on when processing messages in logical order.
- If MQGMO_ALL_SEGMENTS_AVAILABLE is specified without MQGMO_LOGICAL_ORDER, MQGMO_ALL_SEGMENTS_AVAILABLE *always* has an effect. This means that the option must be turned off after the first segment in the logical message has been removed from the queue, in order to be able to remove the remaining segments in the logical message.

If this option is not specified, message segments can be retrieved even when the logical message is incomplete.

While both MQGMO_COMPLETE_MSG and MQGMO_ALL_SEGMENTS_AVAILABLE require all segments to be available before any of them can be retrieved, the former returns the complete message, whereas the latter allows the segments to be retrieved one by one.

If MQGMO_ALL_SEGMENTS_AVAILABLE is specified for a report message, the queue manager checks the queue to see if there is at least one report message for each of the segments that make up the complete logical message. If there is, the MQGMO_ALL_SEGMENTS_AVAILABLE condition is satisfied. However, the queue manager does not check the *type* of the report messages present, and so there might be a mixture of report types in the report messages relating to the segments of the logical message. As a result, the success of MQGMO_ALL_SEGMENTS_AVAILABLE does not imply that MQGMO_COMPLETE_MSG will succeed. If there *is* a mixture of report types present for the segments of a particular logical message, those report messages must be retrieved one by one.

You can specify MQGMO_ALL_SEGMENTS_AVAILABLE with any of the other MQGMO_* options, and with any of the MQMO_* options.

- On z/OS, this option is supported for private and shared queues, but the queue must have an index type of MQIT_GROUP_ID. For shared queues, the CFSTRUCT object that the queue map to must be at CFLEVEL(3) or higher.
- On AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems, this option is supported for all local queues.

Property options: The following options relate to the properties of the message:

MQGMO_PROPERTIES_AS_Q_DEF

Properties of the message, except those contained in the message descriptor (or extension) should be represented as defined by the *PropertyControl* queue attribute. If a *MsgHandle* is provided this option is ignored and the properties of the message are available via the *MsgHandle*, unless the value of the *PropertyControl* queue attribute is MQPROP_FORCE_MQRFH2.

This is the default action if no property options are specified.

MQGMO_PROPERTIES_IN_HANDLE

Properties of the message should be made available via the *MsgHandle*. If no message handle is provided the call fails with reason MQRC_HMSG_ERROR.

Note: If the message is later read by an application that does not create a message handle, the queue manager places any message properties into an MQRFH2 structure. You might find that the presence of an unexpected MQRFH2 header disrupts the behavior of an existing application.

MQGMO_NO_PROPERTIES

No properties of the message, except those contained in the message descriptor (or extension) will be retrieved. If a *MsgHandle* is provided it will be ignored.

MQGMO_PROPERTIES_FORCE_MQRFH2

Properties of the message, except those contained in the message descriptor (or extension) should be represented using MQRFH2 headers. This provides compatibility with earlier version for applications which are expecting to retrieve properties but are unable to be changed to use message handles. If a *MsgHandle* is provided it is ignored.

MQGMO_PROPERTIES_COMPATIBILITY

If the message contains a property with a prefix of "mcd.", "jms.", "usr.", or "mqext.", all message properties are delivered to the application in an MQRFH2 header. Otherwise all properties of the message, except those contained in the message descriptor (or extension), are discarded and are no longer accessible to the application.

Default option: If none of the options described is required, the following option can be used:

MQGMO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values. MQGMO_NONE aids program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

The initial value of the *Options* field is MQGMO_NO_WAIT plus MQGMO_PROPERTIES_AS_Q_DEF.

Reserved1 (MQCHAR)

This is a reserved field. The initial value of this field is a blank character. This field is ignored if *Version* is less than MQGMO_VERSION_2.

Reserved2 (MQLONG)

This is a reserved field. The initial value of this field is a blank character. This field is ignored if *Version* is less than **MQGMO_VERSION_4**.

ResolvedQName (MQCHAR48)

This is an output field that the queue manager sets to the local name of the queue from which the message was retrieved, as defined to the local queue manager. This is different from the name used to open the queue if:

- An alias queue was opened (in which case, the name of the local queue to which the alias resolved is returned), or
- A model queue was opened (in which case, the name of the dynamic local queue is returned).

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ReturnedLength (MQLONG)

This is an output field that the queue manager sets to the length in bytes of the message data returned by the MQGET call in the *Buffer* parameter. If the queue manager does not support this capability, *ReturnedLength* is set to the value MQRL_UNDEFINED.

When messages are converted between encodings or character sets, the message data can sometimes change size. On return from the MQGET call:

- If *ReturnedLength* is not MQRL_UNDEFINED, the number of bytes of message data returned is given by *ReturnedLength*.
- If *ReturnedLength* has the value MQRL_UNDEFINED, the number of bytes of message data returned is usually given by the smaller of *BufferLength* and *DataLength*, but can be *less than* this if the MQGET call completes with reason code MQRC_TRUNCATED_MSG_ACCEPTED. If this happens, the insignificant bytes in the *Buffer* parameter are set to nulls.

The following special value is defined:

MQRL_UNDEFINED

Length of returned data not defined.

On z/OS, the value returned for the *ReturnedLength* field is always MQRL_UNDEFINED.

The initial value of this field is MQRL_UNDEFINED. This field is ignored if *Version* is less than MQGMO_VERSION_3.

Segmentation (MQCHAR)

This is a flag that indicates whether further segmentation is allowed for the message retrieved. It has one of the following values:

MQSEG_INHIBITED

Segmentation not allowed.

MQSEG_ALLOWED

Segmentation allowed.

On z/OS, the queue manager always sets this field to MQSEG_INHIBITED.

This is an output field. The initial value of this field is MQSEG_INHIBITED. This field is ignored if *Version* is less than MQGMO_VERSION_2.

SegmentStatus (MQCHAR)

This is a flag that indicates whether the message retrieved is a segment of a logical message. It has one of the following values:

MQSS_NOT_A_SEGMENT

Message is not a segment.

MQSS_SEGMENT

Message is a segment, but is not the last segment of the logical message.

MQSS_LAST_SEGMENT

Message is the last segment of the logical message.

This is also the value returned if the logical message consists of only one segment.

On z/OS, the queue manager always sets this field to MQSS_NOT_A_SEGMENT.

This is an output field. The initial value of this field is MQSS_NOT_A_SEGMENT. This field is ignored if *Version* is less than MQGMO_VERSION_2.

Signal1 (MQLONG)

This is an input field that is used only in conjunction with the MQGMO_SET_SIGNAL option; it identifies a signal that is to be delivered when a message is available.

Note: The data type and usage of this field are determined by the environment; for this reason, applications that you want to port between different environments must not use signals.

- On z/OS, this field must contain the address of an Event Control Block (ECB). The ECB must be cleared by the application before the MQGET call is issued. The storage containing the ECB must not be freed until the queue is closed. The ECB is posted by the queue manager with one of the signal completion codes described. These completion codes are set in bits 2 through 31 of the ECB, the area defined in the z/OS mapping macro IHAECB as being for a user completion code.
- In all other environments, this is a reserved field; its value is not significant.

The signal completion codes are:

MQEC_MSG_ARRIVED

A suitable message has arrived on the queue. This message has not been reserved for the caller; a second MQGET request must be issued, but another application might retrieve the message before the second request is made.

MQEC_WAIT_INTERVAL_EXPIRED

The specified *WaitInterval* has expired without a suitable message arriving.

MQEC_WAIT_CANCELED

The wait was canceled for an indeterminate reason (such as the queue manager terminating or the queue being disabled). Reissue the request if you want further diagnosis.

MQEC_Q_MGR QUIESCING

The wait was canceled because the queue manager has entered the quiescing state (MQGMO_FAIL_IF QUIESCING was specified on the MQGET call).

MQEC_CONNECTION QUIESCING

The wait was canceled because the connection has entered the quiescing state (MQGMO_FAIL_IF QUIESCING was specified on the MQGET call).

The initial value of this field is determined by the environment:

- On z/OS, the initial value is the null pointer.
- In all other environments, the initial value is 0.

Signal2 (MQLONG)

This is an input field that is used only in conjunction with the MQGMO_SET_SIGNAL option. It is a reserved field; its value is not significant.

The initial value of this field is 0.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQGMO_STRUC_ID

Identifier for get-message options structure.

For the C programming language, the constant MQGMO_STRUC_ID_ARRAY is also defined; this has the same value as MQGMO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQGMO_STRUC_ID.

Version (MQLONG)

Version is the structure version number.

The value must be one of the following:

MQGMO_VERSION_1

Version-1 get-message options structure.

This version is supported in all environments.

MQGMO_VERSION_2

Version-2 get-message options structure.

This version is supported in all environments.

MQGMO_VERSION_3

Version-3 get-message options structure.

This version is supported in all environments.

MQGMO_VERSION_4

Version-4 get-message options structure.

This version is supported in all environments.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQGMO_CURRENT_VERSION

Current version of get-message options structure.

This is always an input field. The initial value of this field is MQGMO_VERSION_1.

WaitInterval (MQLONG)

This is the approximate time, expressed in milliseconds, that the MQGET call waits for a suitable message to arrive (that is, a message satisfying the selection criteria specified in the *MsgDesc* parameter of the MQGET call).

Important: There is no wait, or delay, if a suitable message is available immediately.

See the *MsgId* field described in “MQMD - Message descriptor” on page 396 for more details). If no suitable message has arrived after this time has elapsed, the call completes with MQCC_FAILED and reason code MQRC_NO_MSG_AVAILABLE.

On z/OS, the period of time that the MQGET call actually waits is affected by system loading and work-scheduling considerations, and can vary between the value specified for *WaitInterval* and approximately 100 milliseconds greater than *WaitInterval*.

WaitInterval is used in conjunction with the MQGMO_WAIT or MQGMO_SET_SIGNAL option. It is ignored if neither of these is specified. If one of these is specified, *WaitInterval* must be greater than or equal to zero, or the following special value:

MQWI_UNLIMITED

Unlimited wait interval.

The initial value of this field is 0.

Initial values and language declarations for MQGMO

Table 497. Initial values of fields in MQGMO for MQGMO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQGMO_STRUC_ID	'GMO'
<i>Version</i>	MQGMO_VERSION_1	1
<i>Options</i>	MQGMO_NO_WAIT	0
<i>WaitInterval</i>	None	0
<i>Signal1</i>	None	Null pointer on z/OS ; 0 otherwise
<i>Signal2</i>	None	0
<i>ResolvedQName</i>	None	Null string or blanks
<i>MatchOptions</i>	MQMO_MATCH_MSG_ID + MQMO_MATCH_CORREL_ID	3
<i>GroupStatus</i>	MQGS_NOT_IN_GROUP	' '
<i>SegmentStatus</i>	MQSS_NOT_A_SEGMENT	' '
<i>Segmentation</i>	MQSEG_INHIBITED	' '
<i>Reserved1</i>	None	' '
<i>MsgToken</i>	MQMTOK_NONE	Nulls
<i>ReturnedLength</i>	MQRL_UNDEFINED	-1
<i>Reserved2</i>	None	' '
<i>MsgHandle</i>	MQHM_NONE	0

Table 497. Initial values of fields in MQGMO for MQGMO (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQGMO_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQGMO MyGMO = {MQGMO_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQGMO MQGMO;
struct tagMQGMO {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;           /* Structure version number */
    MQLONG     Options;           /* Options that control the action of */
                                /* MQGET */
    MQLONG     WaitInterval;       /* Wait interval */
    MQLONG     Signal1;           /* Signal */
    MQLONG     Signal2;           /* Signal identifier */
    MQCHAR48    ResolvedQName;     /* Resolved name of destination queue */
    /* Ver:1 */
    MQLONG     MatchOptions;       /* Options controlling selection */
                                /* criteria used for MQGET */
    MQCHAR     GroupStatus;        /* Flag indicating whether message */
                                /* retrieved is in a group */
    MQCHAR     SegmentStatus;     /* Flag indicating whether message */
                                /* retrieved is a segment of a logical */
                                /* message */
    MQCHAR     Segmentation;       /* Flag indicating whether further */
                                /* segmentation is allowed for the */
                                /* message retrieved */
    MQCHAR     Reserved1;         /* Reserved */
    /* Ver:2 */
    MQBYTE16    MsgToken;          /* Message token */
    MQLONG     ReturnedLength;     /* Length of message data returned */
                                /* (bytes) */
    /* Ver:3 */
    MQLONG     Reserved2;         /* Reserved */
    MQHMSG     MsgHandle;         /* Message handle */
    /* Ver:4 */
};
```

- On z/OS, the *Signal1* field is declared as PMQLONG.

COBOL declaration

```
** MQGMO structure
10 MQGMO.
**   Structure identifier
15 MQGMO-STRUCID      PIC X(4).
**   Structure version number
15 MQGMO-VERSION      PIC S9(9) BINARY.
**   Options that control the action of MQGET
15 MQGMO-OPTIONS      PIC S9(9) BINARY.
**   Wait interval
15 MQGMO-WAITINTERVAL PIC S9(9) BINARY.
**   Signal
15 MQGMO-SIGNAL1      PIC S9(9) BINARY.
**   Signal identifier
15 MQGMO-SIGNAL2      PIC S9(9) BINARY.
**   Resolved name of destination queue
15 MQGMO-RESOLVEDQNAME PIC X(48).
**   Options controlling selection criteria used for MQGET
15 MQGMO-MATCHOPTIONS PIC S9(9) BINARY.
**   Flag indicating whether message retrieved is in a group
```

```

15 MQGMO-GROUPSTATUS      PIC X.
**   Flag indicating whether message retrieved is a segment of a
**   logical message
15 MQGMO-SEGMENTSTATUS    PIC X.
**   Flag indicating whether further segmentation is allowed for the
**   message retrieved
15 MQGMO-SEGMENTATION     PIC X.
**   Reserved
15 MQGMO-RESERVED1        PIC X.
**   Message token
15 MQGMO-MSGTOKEN         PIC X(16).
**   Length of message data returned (bytes)
15 MQGMO-RETURNEDLENGTH   PIC S9(9) BINARY.
**   Reserved
15 MQGMO-RESERVED2        PIC S9(9) BINARY.
**   Message handle
15 MQGMO-MSGHANDLE        PIC S9(18) BINARY.

```

- On z/OS, the *Signal1* field is declared as POINTER.

PL/I declaration

```

dcl
  1 MQGMO based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 Options      fixed bin(31), /* Options that control the action of
                                   MQGET */
    3 WaitInterval fixed bin(31), /* Wait interval */
    3 Signal1      fixed bin(31), /* Signal */
    3 Signal2      fixed bin(31), /* Signal identifier */
    3 ResolvedQName char(48),    /* Resolved name of destination
                                   queue */
    3 MatchOptions fixed bin(31), /* Options controlling selection
                                   criteria used for MQGET */
    3 GroupStatus  char(1),      /* Flag indicating whether message
                                   retrieved is in a group */
    3 SegmentStatus char(1),     /* Flag indicating whether message
                                   retrieved is a segment of a logical
                                   message */
    3 Segmentation char(1),     /* Flag indicating whether further
                                   segmentation is allowed for the
                                   message retrieved */
    3 Reserved1    char(1),      /* Reserved */
    3 MsgToken     char(16),     /* Message token */
    3 ReturnedLength fixed bin(31); /* Length of message data returned
                                   (bytes) */
    3 Reserved2    fixed bin(31); /* Reserved */
    3 MsgHandle    fixed bin(63); /* Message handle */

```

- On z/OS, the *Signal1* field is declared as pointer.

High Level Assembler declaration

MQGMO	DSECT		
MQGMO_STRUCID	DS	CL4	Structure identifier
MQGMO_VERSION	DS	F	Structure version number
MQGMO_OPTIONS	DS	F	Options that control the action of MQGET
*			
MQGMO_WAITINTERVAL	DS	F	Wait interval
MQGMO_SIGNAL1	DS	F	Signal
MQGMO_SIGNAL2	DS	F	Signal identifier
MQGMO_RESOLVEDQNAME	DS	CL48	Resolved name of destination queue
MQGMO_MATCHOPTIONS	DS	F	Options controlling selection criteria used for MQGET
*			
MQGMO_GROUPSTATUS	DS	CL1	Flag indicating whether message retrieved is in a group
MQGMO_SEGMENTSTATUS	DS	CL1	Flag indicating whether message retrieved is a segment of a logical message
*			
MQGMO_SEGMENTATION	DS	CL1	Flag indicating whether further segmentation is allowed for the message retrieved
*			
MQGMO_RESERVED1	DS	CL1	Reserved
MQGMO_MSGTOKEN	DS	XL16	Message token
MQGMO_RETURNEDLENGTH	DS	F	Length of message data returned (bytes)
MQGMO_RESERVED2	DS	F	Reserved
MQGMO_MSGHANDLE	DS	D	Message handle

MQGMO_LENGTH	EQU	*-MQGMO
	ORG	MQGMO
MQGMO_AREA	DS	CL(MQGMO_LENGTH)

Visual Basic declaration

```

Type MQGMO
  StrucId      As String*4  'Structure identifier'
  Version      As Long      'Structure version number'
  Options      As Long      'Options that control the action of MQGET'
  WaitInterval As Long      'Wait interval'
  Signal1      As Long      'Signal'
  Signal2      As Long      'Signal identifier'
  ResolvedQName As String*48 'Resolved name of destination queue'
  MatchOptions As Long      'Options controlling selection criteria'
                                'used for MQGET'
  GroupStatus  As String*1  'Flag indicating whether message'
                                'retrieved is in a group'
  SegmentStatus As String*1 'Flag indicating whether message'
                                'retrieved is a segment of a logical'
                                'message'
  Segmentation As String*1  'Flag indicating whether further'
                                'segmentation is allowed for the message'
                                'retrieved'
  Reserved1    As String*1  'Reserved'
  MsgToken     As MQBYTE16  'Message token'
  ReturnedLength As Long    'Length of message data returned (bytes)'
End Type

```

MQIIH - IMS information header

The following table summarizes the fields in the structure.

Table 498. Fields in MQIIH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQIIH structure	StrucLength
<i>Encoding</i>	Reserved	Encoding
<i>CodedCharSetId</i>	Reserved	CodedCharSetId
<i>Format</i>	MQ format name of data that follows MQIIH	Format
<i>Flags</i>	Flags	Flags
<i>LTermOverride</i>	Logical terminal override	LTermOverride
<i>MFSMapName</i>	Message format services map name	MFSMapName
<i>ReplyToFormat</i>	MQ format name of reply message	ReplyToFormat
<i>Authenticator</i>	RACF [®] password or passticket	Authenticator
<i>TranInstanceId</i>	Transaction instance identifier	TranInstanceId
<i>TranState</i>	Transaction state	TranState
<i>CommitMode</i>	Commit mode	CommitMode
<i>SecurityScope</i>	Security scope	SecurityScope
<i>Reserved</i>	Reserved	Reserved

Overview for MQIIH

The MQIIH structure describes the header information for a message sent to IMS across the IMS bridge. For any IBM MQ supported platform you can create and transmit a message that includes the MQIIH structure, but only an IBM MQ for z/OS queue manager can use the IMS bridge. Therefore, for the message to get to IMS from a non-z/OS queue manager, your queue manager network must include at least one z/OS queue manager through which the message can be routed.

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The MQIIH structure describes the information that must be present at the start of a message sent to the IMS bridge through IBM MQ for z/OS.

Format name: MQFMT_IMS.

Character set and encoding: Special conditions apply to the character set and encoding used for the MQIIH structure and application message data:

- Applications that connect to the queue manager that owns the IMS bridge queue must provide an MQIIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQIIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQIIH structure that is in any of the supported character sets and encodings; the receiving message channel agent connected to the queue manager that owns the IMS bridge queue converts the MQIIH.
- The application message data following the MQIIH structure must be in the same character set and encoding as the MQIIH structure. Do not use the *CodedCharSetId* and *Encoding* fields in the MQIIH structure to specify the character set and encoding of the application message data.

You must provide a data-conversion exit to convert the application message data if the data is not one of the built-in formats supported by the queue manager.

Fields for MQIIH

The MQIIH structure contains the following fields; the fields are described in **alphabetical order**:

Authenticator (MQCHAR8)

This is the RACF password or PassTicket. It is optional; if specified, it is used with the user ID in the MQMD security context to build a UTKEN that is sent to IMS to provide a security context. If it is not specified, the user ID is used without verification. This depends on the setting of the RACF switches, which might require an authenticator to be present.

This is ignored if the first byte is blank or null. The following special value can be used:

MQIAUT_NONE

No authentication.

For the C programming language, the constant MQIAUT_NONE_ARRAY is also defined; this has the same value as MQIAUT_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_AUTHENTICATOR_LENGTH. The initial value of this field is MQIAUT_NONE.

CodedCharSetId (MQLONG)

This is a reserved field; its value is not significant. The initial value of this field is 0.

The Character Set Id for supported structures which follow a MQIIH structure is the same as that of the MQIIH structure itself and taken from any preceding MQ header.

CommitMode (MQCHAR)

This is the IMS commit mode. See the *OTMA Reference* for more information about IMS commit modes. The value must be one of the following:

MQICM_COMMIT_THEN_SEND

Commit then send.

This mode implies double queuing of output, but shorter region occupancy times. Fast-path and conversational transactions cannot run with this mode.

MQICM_SEND_THEN_COMMIT

Send then commit.

Any IMS transaction initiated as a result of a commit mode of MQICM_SEND_THEN_COMMIT runs in RESPONSE mode regardless of how the transaction is defined in the IMS system definition (MSGTYPE parameter in the TRANSACT macro). This also applies to transactions initiated by means of a transaction switch.

The initial value of this field is MQICM_COMMIT_THEN_SEND.

Encoding (MQLONG)

This is a reserved field; its value is not significant. The initial value of this field is 0.

The Encoding for supported structures which follow a MQIIH structure is the same as that of the MQIIH structure itself and taken from any preceding MQ header.

Flags (MQLONG)

The flags value must be:

MQIIH_NONE

No flags.

MQIIH_PASS_EXPIRATION

The reply message contains:

- The same expiry report options as the request message
- The remaining expiry time from the request message with no adjustment made for the bridge's processing time

If this value is not set, the expiry time is set to *unlimited*.

MQIIH_REPLY_FORMAT_NONE

Sets the MQIIH.Format field of the reply to MQFMT_NONE.

MQIIH_IGNORE_PURG

Sets the TMAMIPRG indicator in the OTMA prefix, which requests that OTMA ignores PURG calls on the TP PCB for CMO transactions.

MQIIH_CMO_REQUEST_RESPONSE

For Commit Mode 0 (CMO) transactions this flag sets the TMAMHRSP indicator in the OTMA prefix. Setting this indicator requests that OTMA/IMS generate a DFS2082 RESPONSE MODE TRANSACTION TERMINATED WITHOUT REPLY message when the original IMS application program does not reply to the IOPCB nor message switch to another transaction.

The initial value of this field is MQIIH_NONE.

Format (MQCHAR8)

This specifies the MQ format name of the data that follows the MQIIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

LTermOverride (MQCHAR8)

The logical terminal override, placed in the IO PCB field. It is optional; if it is not specified, the TPIPE name is used. It is ignored if the first byte is blank, or null.

The length of this field is given by MQ_LTERM_OVERRIDE_LENGTH. The initial value of this field is 8 blank characters.

MFSMapName (MQCHAR8)

The message format services map name, placed in the IO PCB field. It is optional. On input it represents the MID, on output it represents the MOD. It is ignored if the first byte is blank or null.

The length of this field is given by MQ_MFS_MAP_NAME_LENGTH. The initial value of this field is 8 blank characters.

ReplyToFormat (MQCHAR8)

This is the MQ format name of the reply message that is sent in response to the current message. The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

To convert the data in the reply message using MQGMO_CONVERT, specify either MQIIH.replyToFormat=MQFMT_STRING or MQIIH.replyToFormat=MQFMT_IMS_VAR_STRING. For an explanation of the use of these fields, see [“Format \(MQCHAR8\)” on page 412](#).

If the default value (MQIIH.replyToFormat=MQFMT_NONE) is used on the request message and the reply message is retrieved using MQGMO_CONVERT then no data conversion is performed.

Reserved (MQCHAR)

This is a reserved field; it must be blank.

SecurityScope (MQCHAR)

This indicates the IMS security processing required. The following values are defined:

MQISS_CHECK

Check security scope: an ACEE is built in the control region, but not in the dependent region.

MQISS_FULL

Full security scope: a cached ACEE is built in the control region and a non-cached ACEE is built in the dependent region. If you use MQISS_FULL, ensure that the user ID for which the ACEE is built has access to the resources used in the dependent region.

If neither MQISS_CHECK nor MQISS_FULL is specified for this field, MQISS_CHECK is assumed.

The initial value of this field is MQISS_CHECK.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQIIH_STRUC_ID

Identifier for the IMS information header structure.

For the C programming language, the constant MQIIH_STRUC_ID_ARRAY is also defined; this has the same value as MQIIH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQIIH_STRUC_ID.

StrucLength (MQLONG)

This is the length of MQIIH structure. The value must be:

MQIIH_LENGTH_1

Length of the IMS information header structure.

The initial value of this field is MQIIH_LENGTH_1.

TranInstanceId (MQBYTE16)

This is the transaction instance identifier. This field is used by output messages from IMS, so is ignored on first input. If you set *TranState* to MQITS_IN_CONVERSATION, this must be provided in the next input, and all subsequent inputs, to enable IMS to correlate the messages to the correct conversation. You can use the following special value:

MQITII_NONE

No transaction instance identifier.

For the C programming language, the constant MQITII_NONE_ARRAY is also defined; this has the same value as MQITII_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_TRAN_INSTANCE_ID_LENGTH. The initial value of this field is MQITII_NONE.

TranState (MQCHAR)

This indicates the IMS conversation state. This is ignored on first input because no conversation exists. On subsequent inputs it indicates whether a conversation is active or not. On output it is set by IMS. The value must be one of the following:

MQITS_IN_CONVERSATION


In conversation.

MQITS_NOT_IN_CONVERSATION

Not in conversation.

MQITS_ARCHITECTED

Return transaction state data in architected form.

This value is used only with the IMS /DISPLAY TRAN command. It returns the transaction state data in the IMS architected form instead of character form.  For more information, see [Writing IMS transaction programs through IBM MQ](#).

The initial value of this field is MQITS_NOT_IN_CONVERSATION.

Version (MQLONG)

This is the structure version number. The value must be:

MQIIH_VERSION_1

Version number for IMS information header structure.

The following constant specifies the version number of the current version:

MQIIH_CURRENT_VERSION

Current version of IMS information header structure.

The initial value of this field is MQIIH_VERSION_1.

Initial values and language declarations for MQIIH

Table 499. Initial values of fields in MQIIH for MQIIH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQIIH_STRUC_ID	'IIH?'
<i>Version</i>	MQIIH_VERSION_1	1
<i>StrucLength</i>	MQIIH_LENGTH_1	84
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	None	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQIIH_NONE	0
<i>LTermOverride</i>	None	Blanks
<i>MFSMapName</i>	None	Blanks
<i>ReplyToFormat</i>	MQFMT_NONE	Blanks

Table 499. Initial values of fields in MQIIH for MQIIH (continued)

Field name	Name of constant	Value of constant
<i>Authenticator</i>	MQIAUT_NONE	Blanks
<i>TranInstanceId</i>	MQITII_NONE	Nulls
<i>TranState</i>	MQITS_NOT_IN_CONVERSATION	'?'
<i>CommitMode</i>	MQICM_COMMIT_THEN_SEND	'0'
<i>SecurityScope</i>	MQISS_CHECK	'C'
<i>Reserved</i>	None	'?'

Notes:

1. The symbol ? represents a single blank character.
2. In the C programming language, the macro variable MQIIH_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure:

```
MQIIH MyIIH = {MQIIH_DEFAULT};
```

C declaration

```
typedef struct tagMQIIH MQIIH;
struct tagMQIIH {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     StrucLength;      /* Length of MQIIH structure */
    MQLONG     Encoding;         /* Reserved */
    MQLONG     CodedCharSetId;   /* Reserved */
    MQCHAR8    Format;           /* MQ format name of data that follows
                                MQIIH */
    MQLONG     Flags;            /* Flags */
    MQCHAR8    LTermOverride;    /* Logical terminal override */
    MQCHAR8    MFSMapName;      /* Message format services map name */
    MQCHAR8    ReplyToFormat;    /* MQ format name of reply message */
    MQCHAR8    Authenticator;    /* RACF password or passticket */
    MQBYTE16   TranInstanceId;   /* Transaction instance identifier */
    MQCHAR     TranState;        /* Transaction state */
    MQCHAR     CommitMode;       /* Commit mode */
    MQCHAR     SecurityScope;    /* Security scope */
    MQCHAR     Reserved;         /* Reserved */
};
```

COBOL declaration

```
**  MQIIH structure
10 MQIIH.
**    Structure identifier
15 MQIIH-STRUCID      PIC X(4).
**    Structure version number
15 MQIIH-VERSION      PIC S9(9) BINARY.
**    Length of MQIIH structure
15 MQIIH-STRUCLNGTH  PIC S9(9) BINARY.
**    Reserved
15 MQIIH-ENCODING     PIC S9(9) BINARY.
**    Reserved
15 MQIIH-CODEDCHARSETID PIC S9(9) BINARY.
**    MQ format name of data that follows MQIIH
15 MQIIH-FORMAT       PIC X(8).
**    Flags
15 MQIIH-FLAGS        PIC S9(9) BINARY.
**    Logical terminal override
15 MQIIH-LTERMOVERRIDE PIC X(8).
**    Message format services map name
15 MQIIH-MFSMAPNAME   PIC X(8).
**    MQ format name of reply message
```

```

15 MQIIH-REPLYTOFORMAT PIC X(8).
** RACF password or passticket
15 MQIIH-AUTHENTICATOR PIC X(8).
** Transaction instance identifier
15 MQIIH-TRANINSTANCEID PIC X(16).
** Transaction state
15 MQIIH-TRANSTATE PIC X.
** Commit mode
15 MQIIH-COMMITMODE PIC X.
** Security scope
15 MQIIH-SECURITYSCOPE PIC X.
** Reserved
15 MQIIH-RESERVED PIC X.

```

PL/I declaration

```

dcl
  1 MQIIH based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 StrucLength  fixed bin(31), /* Length of MQIIH structure */
    3 Encoding     fixed bin(31), /* Reserved */
    3 CodedCharSetId fixed bin(31), /* Reserved */
    3 Format        char(8),      /* MQ format name of data that follows
                                   MQIIH */
    3 Flags        fixed bin(31), /* Flags */
    3 LTermOverride char(8),      /* Logical terminal override */
    3 MFSMapName   char(8),      /* Message format services map name */
    3 ReplyToFormat char(8),      /* MQ format name of reply message */
    3 Authenticator char(8),      /* RACF password or passticket */
    3 TranInstanceId char(16),    /* Transaction instance identifier */
    3 TranState    char(1),       /* Transaction state */
    3 CommitMode   char(1),       /* Commit mode */
    3 SecurityScope char(1),      /* Security scope */
    3 Reserved     char(1);      /* Reserved */

```

High Level Assembler declaration

```

MQIIH          DSECT
MQIIH_STRUCID  DS    CL4    Structure identifier
MQIIH_VERSION  DS    F      Structure version number
MQIIH_STRUCLNGTH DS    F      Length of MQIIH structure
MQIIH_ENCODING DS    F      Reserved
MQIIH_CODEDCHARSETID DS    F      Reserved
MQIIH_FORMAT   DS    CL8    MQ format name of data that follows
*              MQIIH
MQIIH_FLAGS    DS    F      Flags
MQIIH_LTERM_OVERRIDE DS    CL8 Logical terminal override
MQIIH_MFSMAPNAME DS    CL8  Message format services map name
MQIIH_REPLYTOFORMAT DS    CL8 MQ format name of reply message
MQIIH_AUTHENTICATOR DS    CL8 RACF password or passticket
MQIIH_TRANINSTANCEID DS    XL16 Transaction instance identifier
MQIIH_TRANSTATE DS    CL1    Transaction state
MQIIH_COMMITMODE DS    CL1    Commit mode
MQIIH_SECURITYSCOPE DS    CL1 Security scope
MQIIH_RESERVED DS    CL1    Reserved
*
MQIIH_LENGTH   EQU    *-MQIIH
ORG    MQIIH
MQIIH_AREA     DS     CL(MQIIH_LENGTH)

```

Visual Basic declaration

```

Type MQIIH
  StrucId      As String*4 'Structure identifier'
  Version      As Long      'Structure version number'
  StrucLength  As Long      'Length of MQIIH structure'
  Encoding     As Long      'Reserved'
  CodedCharSetId As Long    'Reserved'
  Format       As String*8  'MQ format name of data that follows MQIIH'
  Flags       As Long      'Flags'
  LTermOverride As String*8 'Logical terminal override'
  MFSMapName  As String*8  'Message format services map name'
  ReplyToFormat As String*8 'MQ format name of reply message'
  Authenticator As String*8 'RACF password or passticket'
  TranInstanceId As MQBYTE16 'Transaction instance identifier'

```

```

TranState      As String*1 'Transaction state'
CommitMode     As String*1 'Commit mode'
SecurityScope  As String*1 'Security scope'
Reserved       As String*1 'Reserved'
End Type

```

MQIMPO - Inquire message property options

The following table summarizes the fields in the structure. MQIMPO structure - inquire message property options

Table 500. Fields in MQIMPO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options controlling the action of MQINQMP	Options
<i>RequestedEncoding</i>	Encoding into which the inquired property is to be converted	RequestedEncoding
<i>RequestedCCSID</i>	Character set of the inquired property	RequestedCCSID
<i>ReturnedEncoding</i>	Encoding of the returned value	ReturnedEncoding
<i>ReturnedCCSID</i>	Character set of returned value	ReturnedCCSID
<i>Reserved1</i>	Reserved field	ReturnedCCSID
<i>ReturnedName</i>	Name of the inquired property	ReturnedName
<i>TypeString</i>	String representation of the data type of the property	TypeString

Overview for MQIMPO

The inquire message properties options structure.

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The MQIMPO structure allows applications to specify options that control how properties of messages are inquired. The structure is an input parameter on the MQINQMP call.

Character set and encoding: Data in MQIMPO must be in the character set of the application and encoding of the application (MQENC_NATIVE).

Fields for MQIMPO

Inquire message property options structure - fields

The MQIMPO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

Inquire message property options structure - Options field

The following options control the action of MQINQMP. You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

Combinations of options that are not valid are noted; all other combinations are valid.

Value data options: The following options relate to the processing of the value data when the property is retrieved from the message.

MQIMPO_CONVERT_VALUE

This option requests that the value of the property be converted to conform to the *RequestedCCSID* and *RequestedEncoding* values specified before the MQINQMP call returns the property value in the *Value* area.

- If conversion is successful, the *ReturnedCCSID* and *ReturnedEncoding* fields are set to the same as *RequestedCCSID* and *RequestedEncoding* on return from the MQINQMP call.
- If conversion fails, but the MQINQMP call otherwise completes without error, the property value is returned unconverted.

If the property is a string, the *ReturnedCCSID* and *ReturnedEncoding* fields are set to the character set and encoding of the unconverted string.

The completion code is MQCC_WARNING in this case, with reason code MQRC_PROP_VALUE_NOT_CONVERTED. The property cursor is advanced to the returned property.

If the property value expands during conversion, and exceeds the size of the *Value* parameter, the value is returned unconverted, with completion code MQCC_FAILED; the reason code is set to MQRC_PROPERTY_VALUE_TOO_BIG.

The *DataLength* parameter of the MQINQMP call returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

This option also requests that:

- If the property name contains a wildcard, and
- The *ReturnedName* field is initialized with an address or offset for the returned name,

then the returned name is converted to conform to the *RequestedCCSID* and *RequestedEncoding* values.

- If conversion is successful, the *VSCCSID* field of *ReturnedName* and the encoding of the returned name are set to the input value of *RequestedCCSID* and *RequestedEncoding*.
- If conversion fails, but the MQINQMP call otherwise completes without error or warning, the returned name is unconverted. The completion code is MQCC_WARNING in this case, with reason code MQRC_PROP_NAME_NOT_CONVERTED.

The property cursor is advanced to the returned property. MQRC_PROP_VALUE_NOT_CONVERTED is returned if both the value and the name are not converted.

If the returned name expands during conversion, and exceeds the size of the *VSBuFSIZE* field of the *RequestedName*, the returned string is left unconverted, with completion code MQCC_FAILED and the reason code is set to MQRC_PROPERTY_NAME_TOO_BIG.

The *VSLength* field of the MQCHARV structure returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

MQIMPO_CONVERT_TYPE

This option requests that the value of the property be converted from its current data type, into the data type specified on the *Type* parameter of the MQINQMP call.

- If conversion is successful, the *Type* parameter is unchanged on return of the MQINQMP call.
- If conversion fails, but the MQINQMP call otherwise completes without error, the call fails with reason MQRC_PROP_CONV_NOT_SUPPORTED. The property cursor is unchanged.

If the conversion of the data type causes the value to expand during conversion, and the converted value exceeds the size of the *Value* parameter, the value is returned unconverted, with completion code MQCC_FAILED and the reason code is set to MQRC_PROPERTY_VALUE_TOO_BIG.

The *DataLength* parameter of the MQINQMP call returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

If the value of the *Type* parameter of the MQINQMP call is not valid, the call fails with reason MQRC_PROPERTY_TYPE_ERROR.

If the requested data type conversion is not supported, the call fails with reason MQRC_PROP_CONV_NOT_SUPPORTED. The following data type conversions are supported:

Property data type	Supported target data types
MQTYPE_BOOLEAN	MQTYPE_STRING, MQTYPE_INT8, MQTYPE_INT16, MQTYPE_INT32, MQTYPE_INT64
MQTYPE_BYTE_STRING	MQTYPE_STRING
MQTYPE_INT8	MQTYPE_STRING, MQTYPE_INT16, MQTYPE_INT32, MQTYPE_INT64
MQTYPE_INT16	MQTYPE_STRING, MQTYPE_INT32, MQTYPE_INT64
MQTYPE_INT32	MQTYPE_STRING, MQTYPE_INT64
MQTYPE_INT64	MQTYPE_STRING
MQTYPE_FLOAT32	MQTYPE_STRING, MQTYPE_FLOAT64
MQTYPE_FLOAT64	MQTYPE_STRING
MQTYPE_STRING	MQTYPE_BOOLEAN, MQTYPE_INT8, MQTYPE_INT16, MQTYPE_INT32, MQTYPE_INT64, MQTYPE_FLOAT32, MQTYPE_FLOAT64
MQTYPE_NULL	None

The general rules governing the supported conversions are as follows:

- Numeric property values can be converted from one data type to another, provided that no data is lost during the conversion.

For example, the value of a property with data type MQTYPE_INT32 can be converted into a value with data type MQTYPE_INT64, but cannot be converted into a value with data type MQTYPE_INT16.

- A property value of any data type can be converted into a string.
- A string property value can be converted to any other data type provided the string is formatted correctly for the conversion. If an application attempts to convert a string property value that is not formatted correctly, IBM MQ returns reason code MQRC_PROP_NUMBER_FORMAT_ERROR.
- If an application attempts a conversion that is not supported, IBM MQ returns reason code MQRC_PROP_CONV_NOT_SUPPORTED.

The specific rules for converting a property value from one data type to another are as follows:

- When converting an MQTYPE_BOOLEAN property value to a string, the value TRUE is converted to the string "TRUE", and the value false is converted to the string "FALSE".
- When converting an MQTYPE_BOOLEAN property value to a numeric data type, the value TRUE is converted to one, and the value FALSE is converted to zero.
- When converting a string property value to an MQTYPE_BOOLEAN value, the string "TRUE" , or "1" , is converted to TRUE, and the string "FALSE", or "0", is converted to FALSE.

Note that the terms "TRUE" and "FALSE" are not case sensitive.

Any other string cannot be converted; IBM MQ returns reason code MQRC_PROP_NUMBER_FORMAT_ERROR.

- When converting a string property value to a value with data type MQTYPE_INT8, MQTYPE_INT16, MQTYPE_INT32 or MQTYPE_INT64, the string must have the following format:

```
[blanks][sign]digits
```

The meanings of the components of the string are as follows:

blanks

Optional leading blank characters

sign

An optional plus sign (+) or minus sign (-) character.

digits

A contiguous sequence of digit characters (0-9). At least one digit character must be present.

After the sequence of digit characters, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

IBM MQ returns reason code MQRC_PROP_NUMBER_FORMAT_ERROR if the string is not formatted correctly.

- When converting a string property value to a value with data type MQTYPE_FLOAT32 or MQTYPE_FLOAT64, the string must have the following format:

```
[blanks][sign]digits[.digits][e_char[e_sign]e_digits]
```

The meanings of the components of the string are as follows:

blanks

Optional leading blank characters

sign

An optional plus sign (+) or minus sign (-) character.

digits

A contiguous sequence of digit characters (0-9). At least one digit character must be present.

e_char

An exponent character, which is either "E" or "e".

e_sign

An optional plus sign (+) or minus sign (-) character for the exponent.

e_digits

A contiguous sequence of digit characters (0-9) for the exponent. At least one digit character must be present if the string contains an exponent character.

After the sequence of digit characters, or the optional characters representing an exponent, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

IBM MQ returns reason code MQRC_PROP_NUMBER_FORMAT_ERROR if the string is not formatted correctly.

- When converting a numeric property value to a string, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".
- When converting a byte string property value to a string, each byte is converted to the two hexadecimal characters that represent the byte. For example, the byte array {0xF1, 0x12, 0x00, 0xFF} is converted to the string "F11200FF".

MQIMPO_QUERY_LENGTH

Query the type and length of the property value. The length is returned in the *DataLength* parameter of the MQINQMP call. The property value is not returned.

If a *ReturnedName* buffer is specified, the *VSLength* field of the MQCHARV structure is filled in with the length of the property name. The property name is not returned.

Iteration options: The following options relate to iterating over properties, using a name with a wildcard character

MQIMPO_INQ_FIRST

Inquire on the first property that matches the specified name. After this call, a cursor is established on the property that is returned.

This is the default value.

The MQIMPO_INQ_PROP_UNDER_CURSOR option can subsequently be used with an MQINQMP call, if required, to inquire on the same property again.

Note that there is only one property cursor; therefore, if the property name, specified in the MQINQMP call, changes the cursor is reset.

This option is not valid with either of the following options:

MQIMPO_INQ_NEXT
MQIMPO_INQ_PROP_UNDER_CURSOR

MQIMPO_INQ_NEXT

Inquires on the next property that matches the specified name, continuing the search from the property cursor. The cursor is advanced to the property that is returned.

If this is the first MQINQMP call for the specified name, then the first property that matches the specified name is returned.

The MQIMPO_INQ_PROP_UNDER_CURSOR option can subsequently be used with an MQINQMP call if required, to inquire on the same property again.

If the property under the cursor has been deleted, MQINQMP returns the next matching property following the one that has been deleted.

If a property is added that matches the wildcard, while an iteration is in progress, the property might or might not be returned during the completion of the iteration. The property is returned once the iteration restarts using MQIMPO_INQ_FIRST.

A property matching the wildcard that was deleted, while the iteration was in progress, is not returned subsequent to its deletion.

This option is not valid with either of the following options:

MQIMPO_INQ_FIRST
MQIMPO_INQ_PROP_UNDER_CURSOR

MQIMPO_INQ_PROP_UNDER_CURSOR

Retrieve the value of the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired, using either the MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused, when the message handle is specified in the *MsgHandle* field of the MQGMO on an MQGET call, or when the message handle is specified in *OriginalMsgHandle* or *NewMsgHandle* fields of the MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established, or if the property pointed to by the property cursor has been deleted, the call fails with completion code MQCC_FAILED and reason MQRC_PROPERTY_NOT_AVAILABLE.

This option is not valid with either of the following options:

MQIMPO_INQ_FIRST
MQIMPO_INQ_NEXT

If none of the options previously described is required, the following option can be used:

MQIMPO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values.

MQIMPO_NONE aids program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is always an input field. The initial value of this field is MQIMPO_INQ_FIRST.

RequestedCCSID (MQLONG)

Inquire message property options structure - RequestedCCSID field

The character set that the inquired property value is to be converted into if the value is a character string. This is also the character set into which the *ReturnedName* is to be converted when MQIMPO_CONVERT_VALUE or MQIMPO_CONVERT_TYPE is specified.

The initial value of this field is MQCCSI_APPL.

RequestedEncoding (MQLONG)

Inquire message property options structure - RequestedEncoding field

This is the encoding into which the inquired property value is to be converted when MQIMPO_CONVERT_VALUE or MQIMPO_CONVERT_TYPE is specified.

The initial value of this field is MQENC_NATIVE.

Reserved1 (MQCHAR)

This is a reserved field. The initial value of this field is a blank character (4 byte field).

ReturnedCCSID (MQLONG)

Inquire message property options structure - ReturnedCCSID field

On output, this is the character set of the value returned if the *Type* parameter of the MQINQMP call is MQTYPE_STRING.

If the MQIMPO_CONVERT_VALUE option is specified and conversion was successful, the *ReturnedCCSID* field, on return, is the same value as the value passed in.

The initial value of this field is zero.

ReturnedEncoding (MQLONG)

Inquire message property options structure - ReturnedEncoding field

On output, this is the encoding of the value returned.

If the MQIMPO_CONVERT_VALUE option is specified and conversion was successful, the *ReturnedEncoding* field, on return, is the same value as the value passed in.

The initial value of this field is MQENC_NATIVE.

ReturnedName (MQCHARV)

Inquire message property options structure - ReturnedName field

The actual name of the inquired property.

On input a string buffer can be passed in using the *VSPtr* or *VSOffset* field of the MQCHARV structure. The length of the string buffer is specified using the *VSBufsize* field of the MQCHARV structure.

On return from the MQINQMP call, the string buffer is completed with the name of the property that was inquired, provided the string buffer was long enough to fully contain the name. The *VSLength* field of the MQCHARV structure is filled in with the length of the property name. The *VSCCSID* field of the MQCHARV

structure is filled in to indicate the character set of the returned name, whether or not conversion of the name failed.

This is an input/output field. The initial value of this field is MQCHARV_DEFAULT.

StrucId (MQCHAR4)

Inquire message property options structure - StrucId field

This is the structure identifier. The value must be:

MQIMPO_STRUC_ID

Identifier for inquire message property options structure.

For the C programming language, the constant MQIMPO_STRUC_ID_ARRAY is also defined; this has the same value as MQIMPO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQIMPO_STRUC_ID.

TypeString (MQCHAR8)

Inquire message property options structure - TypeString field

A string representation of the data type of the property.

If the property was specified in an MQRFH2 header and the MQRFH2 dt attribute is not recognized, this field can be used to determine the data type of the property. *TypeString* is returned in coded character set 1208 (UTF-8), and is the first eight bytes of the value of the dt attribute of the property that failed to be recognized

This is always an output field. The initial value of this field is the null string in the C programming language, and 8 blank characters in other programming languages.

Version (MQLONG)

Inquire message property options structure - Version field

This is the structure version number. The value must be:

MQIMPO_VERSION_1

Version number for inquire message property options structure.

The following constant specifies the version number of the current version:

MQIMPO_CURRENT_VERSION

Current version of inquire message property options structure.

This is always an input field. The initial value of this field is MQIMPO_VERSION_1.

Initial values and language declarations for MQIMPO

Inquire message property options structure - Initial values

Table 501. Initial values of fields in MQIPMO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQIMPO_STRUC_ID	'IMPO'
<i>Version</i>	MQIMPO_VERSION_1	1
<i>Options</i>	MQIMPO_INQ_FIRST	
<i>RequestedEncoding</i>	MQENC_NATIVE	
<i>RequestedCCSID</i>	MQCCSI_APPL	
<i>ReturnedEncoding</i>	MQENC_NATIVE	
<i>ReturnedCCSID</i>	0	

Table 501. Initial values of fields in MQIPMO (continued)

Field name	Name of constant	Value of constant
<i>Reserved1</i>	0	
<i>ReturnedName</i>	MQCHARV_DEFAULT	
<i>TypeString</i>	Null string or blanks	

Notes:

1. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages.
2. In the C programming language, the macro variable MQIMPO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:

```
MQIMPO MyIMPO = {MQIMPO_DEFAULT};
```

C declaration

Inquire message property options structure - C language declaration

```
typedef struct tagMQIMPO MQIMPO;
struct tagMQIMPO {
    MQCHAR4  StrucId;           /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   Options;          /* Options that control the action of
                               MQINQMP */
    MQLONG   RequestedEncoding; /* Requested encoding of Value */
    MQLONG   RequestedCCSID;    /* Requested character set identifier
                               of Value */
    MQLONG   ReturnedEncoding;  /* Returned encoding of Value */
    MQLONG   ReturnedCCSID;     /* Returned character set identifier
                               of Value */
    MQCHAR   Reserved1;        /* Reserved field */
    MQCHARV  ReturnedName;     /* Returned property name */
    MQCHAR8  TypeString;       /* Property data type as a string */
};
```

COBOL declaration

Inquire message property options structure - COBOL language declaration

```
** MQIMPO structure
10 MQIMPO.
** Structure identifier
15 MQIMPO-STRUCID          PIC X(4).
** Structure version number
15 MQIMPO-VERSION         PIC S9(9) BINARY.
** Options that control the action of MQINQMP
15 MQIMPO-OPTIONS         PIC S9(9) BINARY.
** Requested encoding of VALUE
15 MQIMPO-REQUESTEDENCODING PIC S9(9) BINARY.
** Requested character set identifier of VALUE
15 MQIMPO-REQUESTEDCCSID   PIC S9(9) BINARY.
** Returned encoding of VALUE
15 MQIMPO-RETURNEDENCODING PIC S9(9) BINARY.
** Returned character set identifier of VALUE
15 MQIMPO-RETURNEDCCSID    PIC S9(9) BINARY.
** Reserved field
15 MQIMPO-RESERVED1
** Returned property name
15 MQIMPO-RETURNEDNAME.
** Address of variable length string
20 MQIMPO-RETURNEDNAME-VSPTR  POINTER.
** Offset of variable length string
20 MQIMPO-RETURNEDNAME-VSOFFSET PIC S9(9) BINARY.
** CCSID of variable length string
20 MQIMPO-RETURNEDNAME-VSCCSID PIC S9(9) BINARY.
** Property data type as string
15 MQIMPO-TYPESTRING       PIC S9(9) BINARY.
```

PL/I declaration

Inquire message property options structure - PL/I language declaration

```
dcl
  1 MQIMPO based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31), /* Structure version number */
    3 Options          fixed bin(31), /* Options that control the
                                     action of MQINQMP */
    3 RequestedEncoding fixed bin(31), /* Requested encoding of
                                     Value */
    3 RequestedCCSID   fixed bin(31), /* Requested character set
                                     identifier of Value */
    3 ReturnedEncoding fixed bin(31), /* Returned encoding of
                                     Value */
    3 ReturnedCCSID    fixed bin(31), /* Returned character set
                                     identifier of Value */
    3 Reserved1        fixed bin(31), /* Reserved field */
    3 ReturnedName,    /* Returned property name */
      5 ReturnedName_VSPtr pointer,      /* Address of returned
                                     name */
      5 5 ReturnedName_VSOFFSET fixed bin(31), /* Offset of returned
                                     name */
      5 5 ReturnedName_VSCCSID fixed bin(31), /* CCSID of returned
                                     name */
    3 TypeString       char(8);          /* Property data type as
                                     string */
```

High Level Assembler declaration

Inquire message property options structure - Assembler language declaration

```
MQIMPO          DSECT
MQIMPO_STRUCID   DS    CL4  Structure identifier
MQIMPO_VERSION   DS    F    Structure version number
MQIMPO_OPTIONS   DS    F    Options that control the
*               action of MQINQMP
MQIMPO_REQUESTEDENCODING DS F Requested encoding of VALUE
MQIMPO_REQUESTEDCCSID   DS F Requested character set
*               identifier of VALUE
MQIMPO_RETURNEDENCODING DS F Returned encoding of VALUE
MQIMPO_RETURNEDCCSID   DS F Returned character set
*               identifier of VALUE
MQIMPO_RESERVED1   DS    F    Reserved field
MQIMPO_RETURNEDNAME DS    0F  Force fullword alignment
MQIMPO_RETURNEDNAME_VSPTR DS F Address of returned name
MQIMPO_RETURNEDNAME_VSOFFSET DS F Offset of returned name
MQIMPO_RETURNEDNAME_VSLENGTH DS F Length of returned name
MQIMPO_RETURNEDNAME_VSCCSID DS F CCSID of returned name
MQIMPO_RETURNEDNAME_LENGTH EQU *-MQIMPO_RETURNEDNAME
ORG             MQIMPO_RETURNEDNAME
MQIMPO_RETURNEDNAME_AREA DS    CL(MQIMPO_RETURNEDNAME_LENGTH)
*
MQIMPO_TYPESTRING DS    CL8  Property data type as string
MQIMPO_LENGTH     EQU    *-MQIMPO
MQIMPO_AREA       DS    CL(MQIMPO_LENGTH)
```

MQMD - Message descriptor

The following table summarizes the fields in the structure.

Table 502. Fields in MQMD		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Report</i>	Options for report messages	Report
<i>MsgType</i>	Message type	MsgType
<i>Expiry</i>	Message lifetime	MQMD - Expiry field

Table 502. Fields in MQMD (continued)

Field	Description	Topic
<i>Feedback</i>	Feedback or reason code	MQMD - Feedback field
<i>Encoding</i>	Numeric encoding of message data	Encoding
<i>CodedCharSetId</i>	Character set identifier of message data	CodedCharSetId
<i>Format</i>	Format name of message data	Format
<i>Priority</i>	Message priority	Priority
<i>Persistence</i>	Message persistence	Persistence
<i>MsgId</i>	Message identifier	MQMD - MsgId field
<i>CorrelId</i>	Correlation identifier	CorrelId
<i>BackoutCount</i>	Backout counter	BackoutCount
<i>ReplyToQ</i>	Name of reply queue	ReplyToQ
<i>ReplyToQMgr</i>	Name of reply queue manager	ReplyToQMgr
<i>UserIdentifier</i>	User identifier	UserIdentifier
<i>AccountingToken</i>	Accounting token	AccountingToken
<i>ApplIdentityData</i>	Application data relating to identity	ApplIdentityData
<i>PutApplType</i>	Type of application that put the message	PutApplType
<i>PutApplName</i>	Name of application that put the message	PutApplName
<i>PutDate</i>	Date when message was put	PutDate
<i>PutTime</i>	Time when message was put	PutTime
<i>ApplOriginData</i>	Application data relating to origin	ApplOriginData
Note: The remaining fields are ignored if <i>Version</i> is less than MQMD_VERSION_2.		
<i>GroupId</i>	Group identifier	GroupId
<i>MsgSeqNumber</i>	Sequence number of logical message within group	MsgSeqNumber
<i>Offset</i>	Offset of data in physical message from start of logical message	Offset
<i>MsgFlags</i>	Message flags	MQMD - MsgFlags field
<i>OriginalLength</i>	Length of original message	OriginalLength

Overview for MQMD

Availability: All IBM MQ systems, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQMD structure contains the control information that accompanies the application data when a message travels between the sending and receiving applications. The structure is an input/output parameter on the MQGET, MQPUT, and MQPUT1 calls.

Version: The current version of MQMD is MQMD_VERSION_2. Applications that are intended to be portable between several environments must ensure that the required version of MQMD is supported in all the environments concerned. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQMD that is supported by the environment, but with the initial value of the *Version* field set to MQMD_VERSION_1. To use fields that are not present in the version-1 structure, the application must set the *Version* field to the version number of the version required.

A declaration for the version-1 structure is available with the name MQMD1.

Character set and encoding: Data in MQMD must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE. However, if the application is running as an IBM MQ MQI client, the structure must be in the character set and encoding of the client.

If the sending and receiving queue managers use different character sets or encodings, the data in MQMD is converted automatically. It is not necessary for the application to convert the MQMD.

Using different versions of MQMD: A version-2 MQMD is equivalent to using a version-1 MQMD and prefixing the message data with an MQMDE structure. However, if all the fields in the MQMDE structure have their default values, the MQMDE can be omitted. A version-1 MQMD plus MQMDE are used as described:

- On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *Format* field in MQMD to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager assumes default values for the fields in the MQMDE.

Note: Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on the MQPUT and MQPUT1 calls. However, the queue manager does *not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

- On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a non-default value. The *Format* field in MQMD will have the value MQFMT_MD_EXTENSION to indicate that an MQMDE is present.

The default values that the queue manager uses for the fields in the MQMDE are the same as the initial values of those fields, shown in [Table 507 on page 451](#).

When a message is on a transmission queue, some of the fields in MQMD are set to particular values; see [“MQXQH - Transmission-queue header” on page 592](#) for details.

Message context: Certain fields in MQMD contain the message context. There are two types of message context: *identity context* and *origin context*. Typically:

- Identity context relates to the application that *originally* put the message
- Origin context relates to the application that *most recently* put the message.

These two applications can be the same application, but they can also be different applications (for example, when a message is forwarded from one application to another).

Although identity and origin context typically have the meanings described, the content of both types of context fields in MQMD depends on the MQPMO_*_CONTEXT options that are specified when the message is put. As a result, identity context does not necessarily relate to the application that originally put the message, and origin context does not necessarily relate to the application that most-recently put the message; it depends on the design of the application suite.

The message channel agent (MCA) never alters message context. MCAs that receive messages from remote queue managers use the context option MQPMO_SET_ALL_CONTEXT on the MQPUT or MQPUT1 call. This allows the receiving MCA to preserve exactly the message context that traveled with the message from the sending MCA. However, the result is that the origin context does not relate to either of the MCAs that sent and received the message. The origin context refers to an earlier application that put the message. If all the intermediate applications have passed the message context, the origin context refers to the originating application itself.

In the descriptions, the context fields are described as though they are used as described previously. For more information about message context, see [Message context](#).

Fields for MQMD

The MQMD structure contains the following fields; the fields are described in **alphabetical order**:

Table 503. Fields in MQMD		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Report</i>	Options for report messages	Report
<i>MsgType</i>	Message type	MsgType
<i>Expiry</i>	Message lifetime	MQMD - Expiry field
<i>Feedback</i>	Feedback or reason code	MQMD - Feedback field
<i>Encoding</i>	Numeric encoding of message data	Encoding
<i>CodedCharSetId</i>	Character set identifier of message data	CodedCharSetId
<i>Format</i>	Format name of message data	Format
<i>Priority</i>	Message priority	Priority
<i>Persistence</i>	Message persistence	Persistence
<i>MsgId</i>	Message identifier	MQMD - MsgId field
<i>CorrelId</i>	Correlation identifier	CorrelId
<i>BackoutCount</i>	Backout counter	BackoutCount
<i>ReplyToQ</i>	Name of reply queue	ReplyToQ
<i>ReplyToQMgr</i>	Name of reply queue manager	ReplyToQMgr
<i>UserIdentifier</i>	User identifier	UserIdentifier
<i>AccountingToken</i>	Accounting token	AccountingToken
<i>ApplIdentityData</i>	Application data relating to identity	ApplIdentityData
<i>PutApplType</i>	Type of application that put the message	PutApplType
<i>PutApplName</i>	Name of application that put the message	PutApplName
<i>PutDate</i>	Date when message was put	PutDate
<i>PutTime</i>	Time when message was put	PutTime
<i>ApplOriginData</i>	Application data relating to origin	ApplOriginData
Note: The remaining fields are ignored if <i>Version</i> is less than MQMD_VERSION_2.		
<i>GroupId</i>	Group identifier	GroupId
<i>MsgSeqNumber</i>	Sequence number of logical message within group	MsgSeqNumber
<i>Offset</i>	Offset of data in physical message from start of logical message	Offset
<i>MsgFlags</i>	Message flags	MQMD - MsgFlags field

Table 503. Fields in MQMD (continued)		
Field	Description	Topic
<i>OriginalLength</i>	Length of original message	<u>OriginalLength</u>

AccountingToken (MQBYTE32)

This is the accounting token, part of the **identity context** of the message. For more information about message context, see [“Overview for MQMD” on page 397](#) ; also see [Message context](#).

AccountingToken allows an application to charge appropriately for work done as a result of the message. The queue manager treats this information as a string of bits and does not check its content.

The queue manager generates this information as follows:

- The first byte of the field is set to the length of the accounting information present in the bytes that follow; this length is in the range zero through 30, and is stored in the first byte as a binary integer.
- The second and subsequent bytes (as specified by the length field) are set to the accounting information appropriate to the environment.
 - On z/OS the accounting information is set to:
 - For z/OS batch, the accounting information from the JES JOB card or from a JES ACCT statement in the EXEC card (comma separators are changed to X'FF'). This information is truncated, if necessary, to 31 bytes.
 - For TSO, the user's account number.
 - For CICS, the LU 6.2 unit of work identifier (UEPUOWDS) (26 bytes).
 - For IMS, the 8-character PSB name concatenated with the 16-character IMS recovery token.
 - On IBM i, the accounting information is set to the accounting code for the job.
 - On UNIX systems, the accounting information is set to the numeric user identifier, in ASCII characters.
 - On Windows, the accounting information is set to a Windows security identifier (SID) in a compressed format. The SID uniquely identifies the user identifier stored in the *UserIdentifier* field. When the SID is stored in the *AccountingToken* field, the 6-byte Identifier Authority (located in the third and subsequent bytes of the SID) is omitted. For example, if the Windows SID is 28 bytes long, 22 bytes of SID information are stored in the *AccountingToken* field.
- The last byte (byte 32) of the accounting field is set to the accounting token type (in this case MQACTT_NT_SECURITY_ID, x'0b'):

MQACTT_CICS_LUOW_ID

CICS LUOW identifier.

MQACTT_NT_SECURITY_ID

Windows security identifier.

MQACTT_OS400_ACCOUNT_TOKEN

IBM i accounting token.

MQACTT_UNIX_NUMERIC_ID

UNIX systems numeric identifier.

MQACTT_USER

User-defined accounting token.

MQACTT_UNKNOWN

Unknown accounting-token type.

The accounting-token type is set to an explicit value only in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems. In other environments, the accounting-token type is set to the value MQACTT_UNKNOWN. In these environments use the *PutApplType* field to deduce the type of accounting token received.

- All other bytes are set to binary zero.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If neither MQPMO_SET_IDENTITY_CONTEXT nor MQPMO_SET_ALL_CONTEXT is specified, this field is ignored on input and is an output-only field. For more information about message context, see [Message context](#).

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *AccountingToken* that was transmitted with the message if it was put to a queue. This will be the value of *AccountingToken* that is kept with the message if it is retained (see description of MQPMO_RETAIN in “MQPMO options (MQLONG)” on page 481 for more details about retained publications) but is not used as the *AccountingToken* when the message is sent as a publication to subscribers since they provide a value to override *AccountingToken* in all publications sent to them. If the message has no context, the field is entirely binary zero.

This is an output field for the MQGET call.

This field is not subject to any translation based on the character set of the queue manager; the field is treated as a string of bits, and not as a string of characters.

The queue manager does nothing with the information in this field. The application must interpret the information if it wants to use the information for accounting purposes.

You can use the following special value for the *AccountingToken* field:

MQACT_NONE

No accounting token is specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQACT_NONE_ARRAY is also defined; this has the same value as MQACT_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_ACCOUNTING_TOKEN_LENGTH. The initial value of this field is MQACT_NONE.

ApplIdentityData (MQCHAR32)

This is part of the **identity context** of the message. For more information about message context, see “Overview for MQMD” on page 397 and [Message context](#).

ApplIdentityData is information that is defined by the application suite, and can be used to provide additional information about the message or its originator. The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If a null character is present, the null and any following characters are converted to blanks by the queue manager. If neither MQPMO_SET_IDENTITY_CONTEXT nor MQPMO_SET_ALL_CONTEXT is specified, this field is ignored on input and is an output-only field. For more information about message context, see [Message context](#).

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *ApplIdentityData* that was transmitted with the message if it was put to a queue. This will be the value of *ApplIdentityData* that is kept with the message if it is retained (see description of MQPMO_RETAIN for more details about retained publications) but is not used as the *ApplIdentityData* when the message is sent as a publication to subscribers because they provide a value to override *ApplIdentityData* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_APPL_IDENTITY_DATA_LENGTH. The initial value of this field is the null string in C, and 32 blank characters in other programming languages.

ApplOriginData (MQCHAR4)

This is part of the **origin context** of the message. For more information about message context, see [“Overview for MQMD”](#) on page 397 and [Message context](#).

ApplOriginData is information that is defined by the application suite that can be used to provide additional information about the origin of the message. For example, it could be set by applications running with suitable user authority to indicate whether the identity data is trusted.

The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The queue manager converts the null character and any following characters to blanks. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

This is an output field for the MQGET call. The length of this field is given by MQ_APPL_ORIGIN_DATA_LENGTH. The initial value of this field is the null string in C, and 4 blank characters in other programming languages.

When the message is published, although *ApplOriginData* is set, it is blank in the subscription that it receives.

BackoutCount (MQLONG)

This is a count of the number of times that the message has been previously returned by the MQGET call as part of a unit of work, and subsequently backed out. It helps the application to detect processing errors that are based on message content. The count excludes MQGET calls that specify any of the MQGMO_BROWSE_* options.

The accuracy of this count is affected by the *HardenGetBackout* queue attribute; see [“Attributes for queues”](#) on page 801.

On z/OS, a value of 255 means that the message has been backed out 255 or more times; the value returned is never greater than 255.

This is an output field for the MQGET call. It is ignored for the MQPUT and MQPUT1 calls. The initial value of this field is 0.

CodedCharSetId (MQLONG)

This field specifies the character set identifier of character data within the message body.

Note: Character data in MQMD and the other MQ data structures that are parameters on calls must be in the character set of the queue manager. This is defined by the queue manager's *CodedCharSetId* attribute; see [“Attributes for the queue manager”](#) on page 768 for details of this attribute.

If this field is set to MQCCSI_Q_MGR when calling MQGET with MQGMO_CONVERT in the options, the behavior is different between client and server applications. For server applications, the code page used for character conversion is the *CodedCharSetId* of the queue manager; for client applications, the code page used for character conversion is the current locale code page.

For client applications, MQCCSI_Q_MGR is filled in, based on the locale of the client rather than the one on the queue manager. The exception to that rule is when you put a message to an IMS bridge queue; what is returned, in the *CodedCharSetId* field of MQMD, is the CCSID of the queue manager.

You must not use the following special value:

MQCCSI_APPL

This results in an incorrect value in the *CodedCharSetId* field of the MQMD and causes a return code of MQRC_SOURCE_CCSD_ERROR (or MQRC_FORMAT_ERROR for z/OS) when the message is received using the MQGET call with the MQGMO_CONVERT option.

You can use the following special values:

MQCCSI_Q_MGR

Character data in the message is in the queue manager's character set.

On the MQPUT and MQPUT1 calls, the queue manager changes this value in the MQMD that is sent with the message to the true character set identifier of the queue manager. As a result, the value MQCCSI_Q_MGR is never returned by the MQGET call.

MQCCSI_DEFAULT

The *CodedCharSetId* of the data in the *String* field is defined by the *CodedCharSetId* field in the header structure that precedes the MQCFH structure, or by the *CodedCharSetId* field in the MQMD if the MQCFH is at the start of the message.

MQCCSI_INHERIT

Character data in the message is in the same character set as this structure; this is the queue manager's character set. (For MQMD only, MQCCSI_INHERIT has the same meaning as MQCCSI_Q_MGR).

The queue manager changes this value in the MQMD that is sent with the message to the actual character set identifier of MQMD. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

Do not use MQCCSI_INHERIT if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

MQCCSI_EMBEDDED

Character data in the message is in a character set with the identifier that is contained within the message data itself. There can be any number of character set identifiers embedded within the message data, applying to different parts of the data. This value must be used for PCF messages (with a format of MQFMT_ADMIN, MQFMT_EVENT, or MQFMT_PCF) that contain data in a mixture of character sets. Each MQCFST, MQCFSL, and MQCFSF structure contained within the PCF message must have an explicit character set identifier specified and not MQCCSI_DEFAULT.

If a message of format MQFMT_EMBEDDED_PCF is to contain data in a mixture of character sets, do not use MQCCSI_EMBEDDED. Instead set MQEPH_CCSID_EMBEDDED in the Flags field in the MQEPH structure. This is equivalent to setting MQCCSI_EMBEDDED in the preceding structure. Each MQCFST, MQCFSL, and MQCFSF structure contained within the PCF message must then have an explicit character set identifier specified and not MQCCSI_DEFAULT. For more information on the MQEPH structure, see [“MQEPH - Embedded PCF header” on page 345](#).

Specify this value only on the MQPUT and MQPUT1 calls. If it is specified on the MQGET call, it prevents conversion of the message.

On the MQPUT and MQPUT1 calls, the queue manager changes the values MQCCSI_Q_MGR and MQCCSI_INHERIT in the MQMD that is sent with the message as described above, but does not change the MQMD specified on the MQPUT or MQPUT1 call. No other check is carried out on the value specified.

Applications that retrieve messages must compare this field against the value the application is expecting; if the values differ, the application might need to convert character data in the message.

If you specify the MQGMO_CONVERT option on the MQGET call, this field is an input/output field. The value specified by the application is the coded character set identifier to which to convert the message data if necessary. If conversion is successful or unnecessary, the value is unchanged (except that the value MQCCSI_Q_MGR or MQCCSI_INHERIT is converted to the actual value). If conversion is unsuccessful, the value after the MQGET call represents the coded character set identifier of the unconverted message that is returned to the application.

Otherwise, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQCCSI_Q_MGR.

CorrelId (MQBYTE24)

The CorrelId field is property in the message header that may be used to identify a specific message or group of messages.

This is a byte string that the application can use to relate one message to another, or to relate the message to other work that the application is performing. The correlation identifier is a permanent

property of the message, and persists across restarts of the queue manager. Because the correlation identifier is a byte string and not a character string, the correlation identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 calls, the application can specify any value. The queue manager transmits this value with the message and delivers it to the application that issues the get request for the message.

If the application specifies MQPMO_NEW_CORREL_ID, the queue manager generates a unique correlation identifier which is sent with the message, and also returned to the sending application on output from the MQPUT or MQPUT1 call.

A correlation identifier generated by the queue manager consists of a 3-byte product identifier (AMQ or CSQ in either ASCII or EBCDIC), followed by one reserved byte and a product-specific implementation of a unique string. In IBM MQ this product-specific implementation string contains the first 12 characters of the queue-manager name, and a value derived from the system clock. All queue managers that can intercommunicate must therefore have names that differ in the first 12 characters to ensure that message identifiers are unique. The ability to generate a unique string also depends on the system clock not being changed backward. To eliminate the possibility of a message identifier generated by the queue manager duplicating one generated by the application, the application must avoid generating identifiers with initial characters in the range A through I in ASCII or EBCDIC (X'41' through X'49' and X'C1' through X'C9'). However, the application is not prevented from generating identifiers with initial characters in these ranges.

This generated correlation identifier is kept with the message if it is retained, and is used as the correlation identifier when the message is sent as a publication to subscribers who specify MQCI_NONE in the SubCorrelId field in the MQSD passed on the MQSUB call. See [MQPMO options](#) for more details about retained publications.

When the queue manager or a message channel agent generates a report message, it sets the *CorrelId* field in the way specified by the *Report* field of the original message, either MQRO_COPY_MSG_ID_TO_CORREL_ID or MQRO_PASS_CORREL_ID. Applications that generate report messages must also do this.

For the MQGET call, *CorrelId* is one of the five fields that can be used to select a particular message to be retrieved from the queue. See the description of the *MsgId* field for details of how to specify values for this field.

Specifying MQCI_NONE as the correlation identifier has the same effect as *not* specifying MQMO_MATCH_CORREL_ID, that is, *any* correlation identifier will match.

If the MQGMO_MSG_UNDER_CURSOR option is specified in the *GetMsgOpts* parameter on the MQGET call, this field is ignored.

On return from an MQGET call, the *CorrelId* field is set to the correlation identifier of the message returned (if any).

The following special values can be used:

MQCI_NONE

No correlation identifier is specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQCI_NONE_ARRAY is also defined; this has the same value as MQCI_NONE, but is an array of characters instead of a string.

MQCI_NEW_SESSION

Message is the start of a new session.

This value is recognized by the CICS bridge as indicating the start of a new session, that is, the start of a new sequence of messages.

For the C programming language, the constant MQCI_NEW_SESSION_ARRAY is also defined; this has the same value as MQCI_NEW_SESSION, but is an array of characters instead of a string.

For the MQGET call, this is an input/output field. For the MQPUT and MQPUT1 calls, this is an input field if MQPMO_NEW_CORREL_ID is *not* specified, and an output field if MQPMO_NEW_CORREL_ID is specified. The length of this field is given by MQ_CORREL_ID_LENGTH. The initial value of this field is MQCI_NONE.

Note:

You cannot pass the correlation identifier of a publication in a hierarchy. The field is used by the queue manager.

Encoding (MQLONG)

This specifies the numeric encoding of numeric data in the message; it does not apply to numeric data in the MQMD structure itself. The numeric encoding defines the representation used for binary integers, packed-decimal integers, and floating-point numbers.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that the field is valid. The following special value is defined:

MQENC_NATIVE

The encoding is the default for the programming language and machine on which the application is running.

Note: The value of this constant depends on the programming language and environment. For this reason, applications must be compiled using the header, macro, COPY, or INCLUDE files appropriate to the environment in which the application will run.

Applications that put messages usually specify MQENC_NATIVE. Applications that retrieve messages must compare this field against the value MQENC_NATIVE; if the values differ, the application might need to convert numeric data in the message. Use the MQGMO_CONVERT option to request the queue manager to convert the message as part of the processing of the MQGET call. See [“Machine encodings” on page 863](#) for details of how the *Encoding* field is constructed.

If you specify the MQGMO_CONVERT option on the MQGET call, this field is an input/output field. The value specified by the application is the encoding to which to convert the message data if necessary. If conversion is successful or unnecessary, the value is unchanged. If conversion is unsuccessful, the value after the MQGET call represents the encoding of the unconverted message that is returned to the application.

In other cases, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQENC_NATIVE.

Expiry (MQLONG)

This is a period of time expressed in tenths of a second, set by the application that puts the message. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

For example, to set one minute for the expiry time, you need to set **MQMD.Expiry** to 600.

The value is decremented to reflect the time that the message spends on the destination queue, and also on any intermediate transmission queues if the put is to a remote queue. It can also be decremented by message channel agents to reflect transmission times, if these are significant. Likewise, an application forwarding this message to another queue might decrement the value if necessary, if it has retained the message for a significant time. However, the expiration time is treated as approximate, and the value need not be decremented to reflect small time intervals.

When the message is retrieved by an application using the MQGET call, the *Expiry* field represents the expiry time that still remains.

After a message's expiry time has elapsed, it becomes eligible to be discarded by the queue manager. The message is discarded when a browse or nonbrowse MQGET call occurs that would have returned the message had it not already expired. For example, a nonbrowse MQGET call with the *MatchOptions* field in MQGMO set to MQMO_NONE reading from a FIFO ordered queue discards all the expired messages up to the first unexpired message. With a priority ordered queue, the same call will discard expired messages

of higher priority and messages of an equal priority that arrived on the queue before the first unexpired message.

A message that has expired is never returned to an application (either by a browse or a non-browse MQGET call), so the value in the *Expiry* field of the message descriptor after a successful MQGET call is either greater than zero, or the special value MQEI_UNLIMITED.

If a message is put on a remote queue, the message might expire (and be discarded) while it is on an intermediate transmission queue, before the message reaches the destination queue.

A report is generated when an expired message is discarded, if the message specified one of the MQRO_EXPIRATION_* report options. If none of these options is specified, no such report is generated; the message is assumed to be no longer relevant after this time period (perhaps because a later message has superseded it).

For a message put within syncpoint, the expiry interval begins at the time the message is put, not the time the syncpoint is committed. It is possible that the expiry interval can pass before the syncpoint is committed. In this case the message will be discarded at some time after the commit operation and the message will not be returned to an application in response to an MQGET operation.

Any other program that discards messages based on expiry time must also send an appropriate report message if one was requested.

Notes:

1. If a message is put with an *Expiry* time of zero or a number greater than 999 999 999, the MQPUT or MQPUT1 call fails with reason code MQRC_EXPIRY_ERROR; no report message is generated in this case.

To enable reason code 2013, MQRC_EXPIRY_ERROR, you must enable the environment variable AMQ_ENFORCE_MAX_EXPIRY_ERROR.

The following uses an example for Linux:

```
$ export AMQ_ENFORCE_MAX_EXPIRY_ERROR=True
```

Note that the:

- Important thing is to export the variable
 - Actual value is ignored, however, using True might be helpful when reviewing the setup.
2. Because a message with an expiry time that has elapsed might not be discarded until later, there might be messages on a queue that have passed their expiry time, and that are not therefore eligible for retrieval. These messages nevertheless count toward the number of messages on the queue for all purposes, including depth triggering.
 3. An expiration report is generated, if requested, when the message is discarded, not when it becomes eligible for discarding.
 4. Discarding an expired message, and generating an expiration report if requested, are never part of the application's unit of work, even if the message was scheduled for discarding as a result of an MQGET call operating within a unit of work.
 5. If a nearly-expired message is retrieved by an MQGET call within a unit of work, and the unit of work is subsequently backed out, the message might become eligible to be discarded before it can be retrieved again.
 6. If a nearly-expired message is locked by an MQGET call with MQGMO_LOCK, the message might become eligible to be discarded before it can be retrieved by an MQGET call with MQGMO_MSG_UNDER_CURSOR; reason code MQRC_NO_MSG_UNDER_CURSOR is returned on this subsequent MQGET call if that happens.
 7. When a request message with an expiry time greater than zero is retrieved, the application can take one of the following actions when it sends the reply message:
 - Copy the remaining expiry time from the request message to the reply message.
 - Set the expiry time in the reply message to an explicit value greater than zero.

- Set the expiry time in the reply message to MQEI_UNLIMITED.

The action to take depends on the design of the application. However, the default action for putting messages to a dead-letter (undelivered-message) queue must be to preserve the remaining expiry time of the message, and to continue to decrement it.

8. Trigger messages are always generated with MQEI_UNLIMITED.
9. A message (normally on a transmission queue) that has a *Format* name of MQFMT_XMIT_Q_HEADER has a second message descriptor within the MQXQH. It therefore has two *Expiry* fields associated with it. The following additional points should be noted in this case:
 - When an application puts a message on a remote queue, the queue manager places the message initially on a local transmission queue, and prefixes the application message data with an MQXQH structure. The queue manager sets the values of the two *Expiry* fields to be the same as that specified by the application.

If an application puts a message directly on a local transmission queue, the message data must already begin with an MQXQH structure, and the format name must be MQFMT_XMIT_Q_HEADER. In this case, the application need not set the values of these two *Expiry* fields to be the same. (The queue manager checks that the *Expiry* field within the MQXQH contains a valid value, and that the message data is long enough to include it). For an application that can write directly to the transmission queue, the application has to create a transmission queue header with the embedded message descriptor. However, if the expiry value in the message descriptor written to the transmission queue is inconsistent with the value in the embedded message descriptor, an expiry error rejection occurs.

 - When a message with a *Format* name of MQFMT_XMIT_Q_HEADER is retrieved from a queue (whether this is a normal or a transmission queue), the queue manager decrements *both* these *Expiry* fields with the time spent waiting on the queue. No error is raised if the message data is not long enough to include the *Expiry* field in the MQXQH.
 - The queue manager uses the *Expiry* field in the separate message descriptor (that is, not the one in the message descriptor embedded within the MQXQH structure) to test whether the message is eligible for discarding.
 - If the initial values of the two *Expiry* fields are different, the *Expiry* time in the separate message descriptor when the message is retrieved might be greater than zero (so the message is not eligible for discarding), while the time according to the *Expiry* field in the MQXQH has elapsed. In this case the *Expiry* field in the MQXQH is set to zero.
10. The expiry time on a reply message returned from the IMS bridge is unlimited unless MQIIH_PASS_EXPIRATION is set in the Flags field of the MQIIH. See [Flags](#) for more information.

The following special value is recognized:

MQEI_UNLIMITED

The message has an unlimited expiration time.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQEI_UNLIMITED.

Expired messages on z/OS

On IBM MQ for z/OS, messages that have expired are discarded by the next appropriate MQGET call.

However, if no such call occurs, the expired message is not discarded, and, for some queues, a large number of expired messages can accumulate. To remedy this, set the queue manager to scan queues periodically and discard expired messages on one or more queues in one of the following ways:

Periodic scan

You can specify a period using the EXPRYINT (expiry interval) queue manager attribute. Each time the expiry interval is reached, the queue manager looks for candidate queues that are worth scanning to discard expired messages.

The queue manager maintains information about the expired messages on each queue, and knows whether a scan for expired messages is worthwhile. So, only a selection of queues is scanned at any time.

Shared queues are scanned by only one queue manager in a queue-sharing group. Generally, it is the first queue manager to restart, or the first to have EXPRYINT set. If this queue manager terminates, another queue manager in the queue-sharing group takes over the queue scanning. Set the expiry interval value for all queue managers within a queue-sharing group to the same value.

Note that expiry processing takes place for every queue when a queue manager restarts, regardless of the EXPRYINT setting.

Explicit request

Issue the REFRESH QMGR TYPE(EXPIRY) command, specifying the queue or queues that you want scanned.

V 8.0.0.4 *Enforcing lower expiration times*

Administrators can limit the expiry time of any message put to a queue or topic by using the **CAPEXPY** attribute specified in the **CUSTOM** attribute on the queue or topic.

z/OS To enable this feature on z/OS, you must apply APAR PI50761 and enable the new feature with the MQSC command **RECOVER QMGR (TUNE CAPEXPY ON)**.

Important: The **RECOVER QMGR** command is reserved for use by IBM, and should normally only be used under guidance by IBM support.

z/OS You can either enter the command at a console, or to the queue manager configuration, by adding to a file processed in the CSQINP2 concatenation during queue manager start up. Note that the effect of the command does not persist over a queue manager restart.

An expiry time specified in the **Expiry** field of the MQMD, by an application, which is greater than the **CAPEXPY** value specified in the **CUSTOM** attribute on the queue or topic will be replaced by that **CAPEXPY** value. An expiry time specified by an application, which is lower than the **CAPEXPY** value will be used.

Note that the value of **CAPEXPY** is expressed in tenths of seconds, so one minute has a value of 600.

If more than one object is used on the resolution path, for example, when a message is put to an alias or remote queue, then the lowest of all the **CAPEXPY** values is used as the upper limit for message expiry.

Changes to the **CAPEXPY** values take effect immediately. The expiry value is evaluated for each put to a queue or topic and so is sensitive to the object resolution, which may differ between each put operation.

However, note that existing messages in the queue, prior to a change in **CAPEXPY**, are not affected by the change (that is, their expiry time remains intact). Only new messages that are put into the queue after the change in **CAPEXPY** have the new expiry time.

For example, in a cluster where a put is performed to a queue opened with **MQOO_BIND_NOT_FIXED**, messages can be assigned different expiry values on each put, depending on the **CAPEXPY** value set for the transmission queue, used by the channel, that sends the message to the selected target queue manager.

Note that a put to a queue or topic by a JMS application specifying a delivery delay fails with **MQRC_EXPIRY_ERROR**, if the delivery delay is beyond the resolved expiry time for the target queue or topic. A **CAPEXPY** attribute set on a queue resolved for a JMS Destination can cause this error.

Note: **CAPEXPY** must not be used on any queues that will hold IBM MQ internally-generated messages such as any **SYSTEM.CLUSTER.*** queue and the **SYSTEM.PROTECTION.POLICY.QUEUE**.

Feedback (MQLONG)

The Feedback field is used with a message of type MQMT_REPORT to indicate the nature of the report, and is only meaningful with that type of message.

The field can contain one of the MQFB_* values, or one of the MQRC_* values. Feedback codes are grouped as follows:

MQFB_NONE

No feedback provided.

MQFB_SYSTEM_FIRST

Lowest value for system-generated feedback.

MQFB_SYSTEM_LAST

Highest value for system-generated feedback.

The range of system-generated feedback codes MQFB_SYSTEM_FIRST through MQFB_SYSTEM_LAST includes the general feedback codes listed in this topic (MQFB_*), and also the reason codes (MQRC_*) that can occur when the message cannot be put on the destination queue.

MQFB_APPL_FIRST

Lowest value for application-generated feedback.

MQFB_APPL_LAST

Highest value for application-generated feedback.

Applications that generate report messages must not use feedback codes in the system range (other than MQFB_QUIT), unless they want to simulate report messages generated by the queue manager or message channel agent.

On the MQPUT or MQPUT1 calls, the value specified must either be MQFB_NONE, or be within the system range or application range. This is checked whatever the value of *MsgType*.

General feedback codes:

MQFB_COA

Confirmation of arrival on the destination queue (see MQRO_COA).

MQFB_COD

Confirmation of delivery to the receiving application (see MQRO_COD).

MQFB_EXPIRATION

Message was discarded because it had not been removed from the destination queue before its expiry time had elapsed.

MQFB_PAN

Positive action notification (see MQRO_PAN).

MQFB_NAN

Negative action notification (see MQRO_NAN).

MQFB_QUIT

End application.

This can be used by a workload scheduling program to control the number of instances of an application program that are running. Sending an MQMT_REPORT message with this feedback code to an instance of the application program indicates to that instance that it should stop processing. However, adherence to this convention is a matter for the application; it is not enforced by the queue manager.

Channel feedback codes:

MQFB_CHANNEL_COMPLETED

A channel ended normally.

MQFB_CHANNEL_FAIL

A channel ended abnormally and goes into STOPPED state.

MQFB_CHANNEL_FAIL_RETRY

A channel ended abnormally and goes into RETRY state.

IMS-bridge feedback codes

These codes are used when an unexpected IMS-OTMA sense code is received. The sense code or, when the sense code is 0x1A the reason code associated with that sense code, is indicated in the *Feedback*.

1. For *Feedback* codes in range MQFB_IMS_FIRST (300) through MQFB_IMS_LAST (399), a sense code other than 0x1A was received. The *sense code* is given by the expression (*Feedback* - MQFB_IMS_FIRST+1)
2. For *Feedback* codes in range MQFB_IMS_NACK_1A_REASON_FIRST (600) through MQFB_IMS_NACK_1A_REASON_LAST (855), a sense code of 0x1A was received. The *reason code* associated with the sense code is given by the expression (*Feedback* - MQFB_IMS_NACK_1A_REASON_FIRST)

The meaning of the IMS-OTMA sense codes and corresponding reason codes are described in *Open Transaction Manager Access Guide and Reference*.

The following feedback codes can be generated by the IMS bridge:

MQFB_DATA_LENGTH_ZERO

A segment length was zero in the application data of the message.

MQFB_DATA_LENGTH_NEGATIVE

A segment length was negative in the application data of the message.

MQFB_DATA_LENGTH_TOO_BIG

A segment length was too large in the application data of the message.

MQFB_BUFFER_OVERFLOW

The value of one of the length fields would cause the data to overflow the message buffer.

MQFB_LENGTH_OFF_BY_ONE

The value of one of the length fields was 1 byte too short.

MQFB_IIH_ERROR

The *Format* field in MQMD specifies MQFMT_IMS, but the message does not begin with a valid MQIIH structure.

MQFB_NOT_AUTHORIZED_FOR_IMS

The user ID contained in the message descriptor MQMD, or the password contained in the *Authenticator* field in the MQIIH structure, failed the validation performed by the IMS bridge. As a result the message was not passed to IMS.

MQFB_IMS_ERROR

An unexpected error was returned by IMS. Consult the IBM MQ error log on the system on which the IMS bridge resides for more information about the error.

MQFB_IMS_FIRST

When the IMS-OTMA sense code is not 0x1A, IMS-generated feedback codes are in the range MQFB_IMS_FIRST (300) through MQFB_IMS_LAST (399). The IMS-OTMA sense code itself is *Feedback* minus MQFB_IMS_ERROR.

MQFB_IMS_LAST

Highest value for IMS-generated feedback when the sense code is not 0x1A.

MQFB_IMS_NACK_1A_REASON_FIRST

When the sense code is 0x1A, IMS-generated feedback codes are in the range MQFB_IMS_NACK_1A_REASON_FIRST (600) through MQFB_IMS_NACK_1A_REASON_LAST (855).

MQFB_IMS_NACK_1A_REASON_LAST

Highest value for IMS-generated feedback when the sense code is 0x1A

CICS-bridge feedback codes: The following feedback codes can be generated by the CICS bridge:

MQFB_CICS_APPL_ABENDED

The application program specified in the message abnormally ended. This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_APPL_NOT_STARTED

The EXEC CICS LINK for the application program specified in the message failed. This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_BRIDGE_FAILURE

CICS bridge terminated abnormally without completing normal error processing.

MQFB_CICS_CCSID_ERROR

Character set identifier not valid.

MQFB_CICS_CIH_ERROR

CICS information header structure missing or not valid.

MQFB_CICS_COMMAREA_ERROR

Length of CICS COMMAREA not valid.

MQFB_CICS_CORREL_ID_ERROR

Correlation identifier not valid.

MQFB_CICS_DLQ_ERROR

The CICS bridge task was unable to copy a reply to this request to the dead-letter queue. The request was backed out.

MQFB_CICS_ENCODING_ERROR

Encoding not valid.

MQFB_CICS_INTERNAL_ERROR

CICS bridge encountered an unexpected error.

This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_NOT_AUTHORIZED

User identifier not authorized or password not valid.

This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_UOW_BACKED_OUT

The unit of work was backed out, for one of the following reasons:

- A failure was detected while processing another request within the same unit of work.
- A CICS abend occurred while the unit of work was in progress.

MQFB_CICS_UOW_ERROR

Unit-of-work control field *UOWControl* not valid.

Trace-route message feedback codes:**MQFB_ACTIVITY**

Used with the MQFMT_EMBEDDED_PCF format to allow the option of user data following activity reports.

MQFB_MAX_ACTIVITIES

Returned when the trace-route message is discarded because the number of activities the message has been involved in exceeds the maximum activities limit.

MQFB_NOT_FORWARDED

Returned when the trace-route message is discarded because it is about to be sent to a remote queue manager that does not support trace-route messages.

MQFB_NOT_DELIVERED

Returned when the trace-route message is discarded because it is about to be put on a local queue.

MQFB_UNSUPPORTED_FORWARDING

Returned when the trace-route message is discarded because a value in the forwarding parameter is unrecognized, and is in the rejected bit mask.

MQFB_UNSUPPORTED_DELIVERY

Returned when the trace-route message is discarded because a value in the delivery parameter is unrecognized, and is in the rejected bit mask.

IBM MQ reason codes: For exception report messages, *Feedback* contains an IBM MQ reason code. Among possible reason codes are:

MQRC_PUT_INHIBITED

(2051, X'803') Put calls inhibited for the queue.

MQRC_Q_FULL

(2053, X'805') Queue already contains maximum number of messages.

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_Q_SPACE_NOT_AVAILABLE

(2056, X'808') No space available on disk for queue.

MQRC_PERSISTENT_NOT_ALLOWED

(2048, X'800') Queue does not support persistent messages.

MQRC_MSG_TOO_BIG_FOR_Q_MGR

(2031, X'7EF') Message length greater than maximum for queue manager.

MQRC_MSG_TOO_BIG_FOR_Q

(2030, X'7EE') Message length greater than maximum for queue.

For a full list of reason codes, see:

- For IBM MQ for z/OS, see [API completion and reason codes](#).
- For all other platforms, see [API completion and reason codes](#).

.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is MQFB_NONE.

Format (MQCHAR8)

This is a name that the sender of the message uses to indicate to the receiver the nature of the data in the message. Any characters that are in the character set of the queue manager can be specified for the name, but you must restrict the name to the following:

- Uppercase A through Z
- Numeric digits 0 through 9

If other characters are used, it might not be possible to translate the name between the character sets of the sending and receiving queue managers.

Pad the name with blanks to the length of the field, or use a null character to terminate the name before the end of the field; the null and any subsequent characters are treated as blanks. Do not specify a name with leading or embedded blanks. For the MQGET call, the queue manager returns the name padded with blanks to the length of the field.

The queue manager does not check that the name complies with the recommendations described above.

Names beginning MQ in upper, lower, and mixed case have meanings that are defined by the queue manager; do not use names beginning with these letters for your own formats. The queue manager built-in formats are:

MQFMT_NONE

The nature of the data is undefined: the data cannot be converted when the message is retrieved from a queue using the MQGMO_CONVERT option.

If you specify MQGMO_CONVERT on the MQGET call, and the character set or encoding of data in the message differs from that specified in the *MsgDesc* parameter, the message is returned with the following completion and reason codes (assuming no other errors):

- Completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR if the MQFMT_NONE data is at the beginning of the message.

- Completion code MQCC_OK and reason code MQRC_NONE if the MQFMT_NONE data is at the end of the message (that is, preceded by one or more MQ header structures). The MQ header structures are converted to the requested character set and encoding in this case.

For the C programming language, the constant MQFMT_NONE_ARRAY is also defined; this has the same value as MQFMT_NONE, but is an array of characters instead of a string.

MQFMT_ADMIN

The message is a command-server request or reply message in programmable command format (PCF). Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call. See [Using Programmable Command Formats](#) for more information about using programmable command format messages.

For the C programming language, the constant MQFMT_ADMIN_ARRAY is also defined; this has the same value as MQFMT_ADMIN, but is an array of characters instead of a string.

MQFMT_CICS

The message data begins with the CICS information header MQCIH, followed by the application data. The format name of the application data is given by the *Format* field in the MQCIH structure.

On z/OS, specify the MQGMO_CONVERT option on the MQGET call to convert messages that have format MQFMT_CICS.

For the C programming language, the constant MQFMT_CICS_ARRAY is also defined; this has the same value as MQFMT_CICS, but is an array of characters instead of a string.

MQFMT_COMMAND_1

The message is an MQSC command-server reply message containing the object count, completion code, and reason code. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_COMMAND_1_ARRAY is also defined; this has the same value as MQFMT_COMMAND_1, but is an array of characters instead of a string.

MQFMT_COMMAND_2

The message is an MQSC command-server reply message containing information about the objects requested. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_COMMAND_2_ARRAY is also defined; this has the same value as MQFMT_COMMAND_2, but is an array of characters instead of a string.

MQFMT_DEAD_LETTER_HEADER

The message data begins with the dead-letter header MQDLH. The data from the original message immediately follows the MQDLH structure. The format name of the original message data is given by the *Format* field in the MQDLH structure; see [“MQDLH - Dead-letter header”](#) on page 333 for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

COA and COD reports are not generated for messages that have a *Format* of MQFMT_DEAD_LETTER_HEADER.

For the C programming language, the constant MQFMT_DEAD_LETTER_HEADER_ARRAY is also defined; this has the same value as MQFMT_DEAD_LETTER_HEADER, but is an array of characters instead of a string.

MQFMT_DIST_HEADER

The message data begins with the distribution-list header MQDH; this includes the arrays of MQOR and MQPMR records. The distribution-list header can be followed by additional data. The format of the additional data (if any) is given by the *Format* field in the MQDH structure; see [“MQDH - Distribution header”](#) on page 327 for details of this structure. Messages with format MQFMT_DIST_HEADER can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

This format is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

For the C programming language, the constant MQFMT_DIST_HEADER_ARRAY is also defined; this has the same value as MQFMT_DIST_HEADER, but is an array of characters instead of a string.

MQFMT_EMBEDDED_PCF

Format for a trace-route message, provided that the PCF command value is set to MQCMD_TRACE_ROUTE. Using this format allows user data to be sent along with the trace-route message, provided that their applications can cope with preceding PCF parameters.

The PCF header **must** be the first header, or the message will not be treated as a trace-route message. This means that the message cannot be in a group, and that trace-route messages cannot be segmented. If a trace-route message is sent in a group the message is rejected with reason code MQRC_MSG_NOT_ALLOWED_IN_GROUP.

Note that MQFMT_ADMIN can also be used for the format of a trace-route message, but in this case no user data can be sent along with the trace-route message.

MQFMT_EVENT

The message is an MQ event message that reports an event that occurred. Event messages have the same structure as programmable commands; see [PCF command messages](#) for more information about this structure, and [Event monitoring](#) for information about events.

Version-1 event messages can be converted in all environments if the MQGMO_CONVERT option is specified on the MQGET call. Version-2 event messages can be converted only on z/OS.

For the C programming language, the constant MQFMT_EVENT_ARRAY is also defined; this has the same value as MQFMT_EVENT, but is an array of characters instead of a string.

MQFMT_IMS

The message data begins with the IMS information header MQIIH, which is followed by the application data. The format name of the application data is given by the *Format* field in the MQIIH structure.

For details of how MQIIH structure is handled when using MQGET with MQGMO_CONVERT, see [“Format \(MQCHAR8\)” on page 383](#) and [“ReplyToFormat \(MQCHAR8\)” on page 384](#).

For the C programming language, the constant MQFMT_IMS_ARRAY is also defined; this has the same value as MQFMT_IMS, but is an array of characters instead of a string.

MQFMT_IMS_VAR_STRING

The message is an IMS variable string, which is a string of the form 11zzccc, where:

11

is a 2-byte length field specifying the total length of the IMS variable string item. This length is equal to the length of 11 (2 bytes), plus the length of zz (2 bytes), plus the length of the character string itself. 11 is a 2-byte binary integer in the encoding specified by the *Encoding* field.

zz

is a 2-byte field containing flags that are significant to IMS. zz is a byte string consisting of two MQBYTE fields, and is transmitted without change from sender to receiver (that is, zz is not subject to any conversion).

ccc

is a variable-length character string containing 11-4 characters. ccc is in the character set specified by the *CodedCharSetId* field.

On z/OS, the message data can consist of a sequence of IMS variable strings butted together, with each string being of the form 11zzccc. There must be no bytes skipped between successive IMS variable strings. This means that if the first string has an odd length, the second string will be misaligned, that is, it will not begin on a boundary that is a multiple of two. Take care when constructing such strings on machines that require alignment of elementary data types.

Use the MQGMO_CONVERT option on the MQGET call to convert messages that have format MQFMT_IMS_VAR_STRING.

For the C programming language, the constant MQFMT_IMS_VAR_STRING_ARRAY is also defined; this has the same value as MQFMT_IMS_VAR_STRING, but is an array of characters instead of a string.

MQFMT_MD_EXTENSION

The message data begins with the message-descriptor extension MQMDE, and is optionally followed by other data (usually the application message data). The format name, character set, and encoding of the data that follow the MQMDE are given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMDE. See [“MQMDE - Message descriptor extension” on page 447](#) for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_MD_EXTENSION_ARRAY is also defined; this has the same value as MQFMT_MD_EXTENSION, but is an array of characters instead of a string.

MQFMT_PCF

The message is a user-defined message that conforms to the structure of a programmable command format (PCF) message. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call. See [Using Programmable Command Formats](#) for more information about using programmable command format messages.

For the C programming language, the constant MQFMT_PCF_ARRAY is also defined; this has the same value as MQFMT_PCF, but is an array of characters instead of a string.

MQFMT_REF_MSG_HEADER

The message data begins with the reference message header MQRMH, and is optionally followed by other data. The format name, character set, and encoding of the data is given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQRMH. See [“MQRMH - Reference message header” on page 521](#) for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

This format is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

For the C programming language, the constant MQFMT_REF_MSG_HEADER_ARRAY is also defined; this has the same value as MQFMT_REF_MSG_HEADER, but is an array of characters instead of a string.

MQFMT_RF_HEADER

The message data begins with the rules and formatting header MQRFH, and is optionally followed by other data. The format name, character set, and encoding of the data (if any) are given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQRFH. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_RF_HEADER_ARRAY is also defined; this has the same value as MQFMT_RF_HEADER, but is an array of characters instead of a string.

MQFMT_RF_HEADER_2

The message data begins with the version-2 rules and formatting header MQRFH2, and is optionally followed by other data. The format name, character set, and encoding of the optional data (if any) are given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQRFH2. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_RF_HEADER_2_ARRAY is also defined; this has the same value as MQFMT_RF_HEADER_2, but is an array of characters instead of a string.

MQFMT_STRING

The application message data can be either an SBCS string (single-byte character set), or a DBCS string (double-byte character set). Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_STRING_ARRAY is also defined; this has the same value as MQFMT_STRING, but is an array of characters instead of a string.

MQFMT_TRIGGER

The message is a trigger message, described by the MQTM structure; see [“MQTM - Trigger message” on page 572](#) for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_TRIGGER_ARRAY is also defined; this has the same value as MQFMT_TRIGGER, but is an array of characters instead of a string.

MQFMT_WORK_INFO_HEADER

The message data begins with the work information header MQWIIH, which is followed by the application data. The format name of the application data is given by the *Format* field in the MQWIIH structure.

On z/OS, specify the MQGMO_CONVERT option on the MQGET call to convert the *user data* in messages that have format MQFMT_WORK_INFO_HEADER. However, the MQWIIH structure itself is always returned in the queue-manager's character set and encoding (that is, the MQWIIH structure is converted whether or not the MQGMO_CONVERT option is specified).

For the C programming language, the constant MQFMT_WORK_INFO_HEADER_ARRAY is also defined; this has the same value as MQFMT_WORK_INFO_HEADER, but is an array of characters instead of a string.

MQFMT_XMIT_Q_HEADER

The message data begins with the transmission queue header MQXQH. The data from the original message immediately follows the MQXQH structure. The format name of the original message data is given by the *Format* field in the MQMD structure, which is part of the transmission queue header MQXQH. See [“MQXQH - Transmission-queue header”](#) on page 592 for details of this structure.

COA and COD reports are not generated for messages that have a *Format* of MQFMT_XMIT_Q_HEADER.

For the C programming language, the constant MQFMT_XMIT_Q_HEADER_ARRAY is also defined; this has the same value as MQFMT_XMIT_Q_HEADER, but is an array of characters instead of a string.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

GroupId (MQBYTE24)

This is a byte string that is used to identify the particular message group or logical message to which the physical message belongs. *GroupId* is also used if segmentation is allowed for the message. In all these cases, *GroupId* has a non-null value, and one or more of the following flags is set in the *MsgFlags* field:

- MQMF_MSG_IN_GROUP
- MQMF_LAST_MSG_IN_GROUP
- MQMF_SEGMENT
- MQMF_LAST_SEGMENT
- MQMF_SEGMENTATION_ALLOWED

If none of these flags is set, *GroupId* has the special null value MQGI_NONE.

The application does not need to set this field on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_GROUP_ID is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *GroupId* is set to an appropriate value.

Message groups and segments can be processed correctly only if the group identifier is unique. For this reason, *applications must not generate their own group identifiers* ; instead, applications must do one of the following:

- If MQPMO_LOGICAL_ORDER is specified, the queue manager automatically generates a unique group identifier for the first message in the group or segment of the logical message, and uses that group identifier for the remaining messages in the group or segments of the logical message, so the application does not need to take any special action. This is the recommended procedure.

- If MQPMO_LOGICAL_ORDER is *not* specified, the application must request the queue manager to generate the group identifier, by setting *GroupId* to MQGI_NONE on the first MQPUT or MQPUT1 call for a message in the group or segment of the logical message. The group identifier returned by the queue manager on output from that call must then be used for the remaining messages in the group or segments of the logical message. If a message group contains segmented messages, the same group identifier must be used for all segments and messages in the group.

When MQPMO_LOGICAL_ORDER is not specified, messages in groups and segments of logical messages can be put in any order (for example, in reverse order), but the group identifier must be allocated by the *first* MQPUT or MQPUT1 call that is issued for any of those messages.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value described in [Physical order on a queue](#). On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message if the object opened is a single queue and not a distribution list, but leaves it unchanged if the object opened is a distribution list. In the latter case, if the application needs to know the group identifiers generated, the application must provide MQPMR records containing the *GroupId* field.

On input to the MQGET call, the queue manager uses the value described in [Table 495 on page 369](#). On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The following special value is defined:

MQGI_NONE

No group identifier specified.

The value is binary zero for the length of the field. This is the value that is used for messages that are not in groups, not segments of logical messages, and for which segmentation is not allowed.

For the C programming language, the constant MQGI_NONE_ARRAY is also defined; this has the same value as MQGI_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_GROUP_ID_LENGTH. The initial value of this field is MQGI_NONE. This field is ignored if *Version* is less than MQMD_VERSION_2.

MsgFlags (MQLONG)

MsgFlags are flags that specify attributes of the message, or control its processing.

MsgFlags are divided into the following categories:

- Segmentation flags
- Status flags

Segmentation flags: When a message is too big for a queue, an attempt to put the message on the queue typically fails. Segmentation is a technique whereby the queue manager or application splits the message into smaller pieces called segments, and places each segment on the queue as a separate physical message. The application that retrieves the message can either retrieve the segments one by one, or request the queue manager to reassemble the segments into a single message that is returned by the MQGET call. The latter is achieved by specifying the MQGMO_COMPLETE_MSG option on the MQGET call, and supplying a buffer that is big enough to accommodate the complete message. (See “MQGMO - Get-message options” on page 350 for details of the MQGMO_COMPLETE_MSG option.) A message can be segmented at the sending queue manager, at an intermediate queue manager, or at the destination queue manager.

You can specify one of the following to control the segmentation of a message:

MQMF_SEGMENTATION_INHIBITED

This option prevents the message being broken into segments by the queue manager. If specified for a message that is already a segment, this option prevents the segment being broken into smaller segments.

The value of this flag is binary zero. This is the default.

MQMF_SEGMENTATION_ALLOWED

This option allows the message to be broken into segments by the queue manager. If specified for a message that is already a segment, this option allows the segment to be broken into smaller segments. MQMF_SEGMENTATION_ALLOWED can be set without either MQMF_SEGMENT or MQMF_LAST_SEGMENT being set.

- On z/OS, the queue manager does not support the segmentation of messages. If a message is too big for the queue, the MQPUT or MQPUT1 call fails with reason code MQRC_MSG_TOO_BIG_FOR_Q. However, the MQMF_SEGMENTATION_ALLOWED option can still be specified, and allows the message to be segmented at a remote queue manager.

When the queue manager segments a message, the queue manager turns on the MQMF_SEGMENT flag in the copy of the MQMD that is sent with each segment, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call. For the last segment in the logical message, the queue manager also turns on the MQMF_LAST_SEGMENT flag in the MQMD that is sent with the segment.

Note: Take care when putting messages with MQMF_SEGMENTATION_ALLOWED but without MQPMO_LOGICAL_ORDER. If the message is:

- Not a segment, and
- Not in a group, and
- Not being forwarded,

the application must reset the *GroupId* field to MQGI_NONE before *each* MQPUT or MQPUT1 call, so that the queue manager can generate a unique group identifier for each message. If this is not done, unrelated messages can have the same group identifier, which might lead to incorrect processing subsequently. See the descriptions of the *GroupId* field and the MQPMO_LOGICAL_ORDER option for more information about when to reset the *GroupId* field.

The queue manager splits messages into segments as necessary so that the segments (plus any required header data) fit on the queue. However, there is a lower limit for the size of a segment generated by the queue manager, and only the last segment created from a message can be smaller than this limit (the lower limit for the size of an application-generated segment is one byte). Segments generated by the queue manager might be of unequal length. The queue-manager processes the message as follows:

- User-defined formats are split on boundaries that are multiples of 16 bytes; the queue manager does not generate segments that are smaller than 16 bytes (other than the last segment).
- Built-in formats other than MQFMT_STRING are split at points appropriate to the nature of the data present. However, the queue manager never splits a message in the middle of an IBM MQ header structure. This means that a segment containing a single MQ header structure cannot be split further by the queue manager, and as a result the minimum possible segment size for that message is greater than 16 bytes.

The second or later segment generated by the queue manager begins with one of the following:

- An MQ header structure
- The start of the application message data
- Part of the way through the application message data
- MQFMT_STRING is split without regard for the nature of the data present (SBCS, DBCS, or mixed SBCS/DBCS). When the string is DBCS or mixed SBCS/DBCS, this might result in segments that cannot be converted from one character set to another. The queue manager never splits MQFMT_STRING messages into segments that are smaller than 16 bytes (other than the last segment).
- The queue manager sets the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMD of each segment to describe correctly the data present at the *start* of the segment; the format name is either the name of a built-in format, or the name of a user-defined format.

- The *Report* field in the MQMD of segments with *Offset* greater than zero is modified. For each report type, if the report option is MQRO_*_WITH_DATA, but the segment cannot contain any of the first 100 bytes of user data (that is, the data following any IBM MQ header structures that may be present), the report option is changed to MQRO_*.

The queue manager follows the above rules, but otherwise splits messages unpredictably; do not make assumptions about where a message is split.

For *persistent* messages, the queue manager can perform segmentation only within a unit of work:

- If the MQPUT or MQPUT1 call is operating within a user-defined unit of work, that unit of work is used. If the call fails during the segmentation process, the queue manager removes any segments that were placed on the queue as a result of the failing call. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work just for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically. If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work exists, the queue manager cannot perform segmentation. If the message does not require segmentation, the call can still succeed. But if the message requires segmentation, the call fails with reason code MQRC_UOW_NOT_AVAILABLE.

For *nonpersistent* messages, the queue manager does not require a unit of work to be available in order to perform segmentation.

Take special care when converting data in messages that might be segmented:

- If the receiving application converts data on the MQGET call, and specifies the MQGMO_COMPLETE_MSG option, the data-conversion exit is passed the complete message for the exit to convert, and the fact that the message was segmented is apparent to the exit.
- If the receiving application retrieves one segment at a time, the data-conversion exit is invoked to convert one segment at a time. The exit must therefore convert the data in a segment independently of the data in any of the other segments.

If the nature of the data in the message is such that arbitrary segmentation of the data on 16-byte boundaries might result in segments that cannot be converted by the exit, or the format is MQFMT_STRING and the character set is DBCS or mixed SBCS/DBCS, the sending application must create and put the segments, specifying MQMF_SEGMENTATION_INHIBITED to suppress further segmentation. In this way, the sending application can ensure that each segment contains sufficient information to allow the data-conversion exit to convert the segment successfully.

- If sender conversion is specified for a sending message channel agent (MCA), the MCA converts only messages that are not segments of logical messages; the MCA never attempts to convert messages that are segments.

This flag is an input flag on the MQPUT and MQPUT1 calls, and an output flag on the MQGET call. On the latter call, the queue manager also echoes the value of the flag to the *Segmentation* field in MQGMO.

The initial value of this flag is MQMF_SEGMENTATION_INHIBITED.

Status flags: These are flags that indicate whether the physical message belongs to a message group, is a segment of a logical message, both, or neither. One or more of the following can be specified on the MQPUT or MQPUT1 call, or returned by the MQGET call:

MQMF_MSG_IN_GROUP

Message is a member of a group.

MQMF_LAST_MSG_IN_GROUP

Message is the last logical message in a group.

If this flag is set, the queue manager turns on MQMF_MSG_IN_GROUP in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

It is valid for a group to consist of only one logical message. If this is the case, MQMF_LAST_MSG_IN_GROUP is set, but the *MsgSeqNumber* field has the value one.

MQMF_SEGMENT

Message is a segment of a logical message.

When MQMF_SEGMENT is specified without MQMF_LAST_SEGMENT, the length of the application message data in the segment (*excluding* the lengths of any IBM MQ header structures that might be present) must be at least one. If the length is zero, the MQPUT or MQPUT1 call fails with reason code MQRC_SEGMENT_LENGTH_ZERO.

On z/OS, this option is not supported if the message is being put on a queue that has an index type of MQIT_GROUP_ID.

MQMF_LAST_SEGMENT

Message is the last segment of a logical message.

If this flag is set, the queue manager turns on MQMF_SEGMENT in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

A logical message can consist of only one segment. If so, MQMF_LAST_SEGMENT is set, but the *Offset* field has the value zero.

When MQMF_LAST_SEGMENT is specified, the length of the application message data in the segment (*excluding* the lengths of any header structures that might be present) can be zero.

On z/OS, this option is not supported if the message is being put on a queue that has an index type of MQIT_GROUP_ID.

The application must ensure that these flags are set correctly when putting messages. If MQPMO_LOGICAL_ORDER is specified, or was specified on the preceding MQPUT call for the queue handle, the settings of the flags must be consistent with the group and segment information retained by the queue manager for the queue handle. The following conditions apply to *successive* MQPUT calls for the queue handle when MQPMO_LOGICAL_ORDER is specified:

- If there is no current group or logical message, all these flags (and combinations of them) are valid.
- Once MQMF_MSG_IN_GROUP has been specified, it must remain on until MQMF_LAST_MSG_IN_GROUP is specified. The call fails with reason code MQRC_INCOMPLETE_GROUP if this condition is not satisfied.
- Once MQMF_SEGMENT has been specified, it must remain on until MQMF_LAST_SEGMENT is specified. The call fails with reason code MQRC_INCOMPLETE_MSG if this condition is not satisfied.
- Once MQMF_SEGMENT has been specified without MQMF_MSG_IN_GROUP, MQMF_MSG_IN_GROUP must remain *off* until after MQMF_LAST_SEGMENT has been specified. The call fails with reason code MQRC_INCOMPLETE_MSG if this condition is not satisfied.

Physical order on a queue shows the valid combinations of the flags, and the values used for various fields.

These flags are input flags on the MQPUT and MQPUT1 calls, and output flags on the MQGET call. On the latter call, the queue manager also echoes the values of the flags to the *GroupStatus* and *SegmentStatus* fields in MQGMO.

You cannot use grouped or segmented messages with Publish/Subscribe.

Default flags: The following can be specified to indicate that the message has default attributes:

MQMF_NONE

No message flags (default message attributes).

This inhibits segmentation, and indicates that the message is not in a group and is not a segment of a logical message. MQMF_NONE is defined to aid program documentation. It is not intended that this flag be used with any other, but as its value is zero, such use cannot be detected.

The *MsgFlags* field is partitioned into subfields; for details see [“Report options and message flags” on page 866](#).

The initial value of this field is MQMF_NONE. This field is ignored if *Version* is less than MQMD_VERSION_2.

MsgId (MQBYTE24)

This is a byte string that is used to distinguish one message from another. Generally, no two messages should have the same message identifier, although this is not disallowed by the queue manager. The message identifier is a permanent property of the message, and persists across restarts of the queue manager. Because the message identifier is a byte string and not a character string, the message identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 calls, if MQMI_NONE or MQPMO_NEW_MSG_ID is specified by the application, the queue manager generates a unique message identifier² when the message is put, and places it in the message descriptor sent with the message. The queue manager also returns this message identifier in the message descriptor belonging to the sending application. The application can use this value to record information about particular messages, and to respond to queries from other parts of the application.

If the message is being put to a topic, the queue manager generates unique message identifiers as necessary for each message published. If MQPMO_NEW_MSG_ID is specified by the application, the queue manager generates a unique message identifier to return on output. If MQMI_NONE is specified by the application, the value of the *MsgId* field in the MQMD is unchanged on return from the call.

See the description of MQPMO_RETAIN in [“MQPMO options \(MQLONG\)” on page 481](#) for more details about retained publications.

If the message is being put to a distribution list, the queue manager generates unique message identifiers as necessary, but the value of the *MsgId* field in MQMD is unchanged on return from the call, even if MQMI_NONE or MQPMO_NEW_MSG_ID was specified. If the application needs to know the message identifiers generated by the queue manager, the application must provide MQPMR records containing the *MsgId* field.

The sending application can also specify a value for the message identifier other than MQMI_NONE; this stops the queue manager generating a unique message identifier. An application that is forwarding a message can use this to propagate the message identifier of the original message.

The queue manager does not use this field except to:

- Generate a unique value if requested, as described above
- Deliver the value to the application that issues the get request for the message
- Copy the value to the *CorrelId* field of any report message that it generates about this message (depending on the *Report* options)

When the queue manager or a message channel agent generates a report message, it sets the *MsgId* field in the way specified by the *Report* field of the original message, either MQRO_NEW_MSG_ID or MQRO_PASS_MSG_ID. Applications that generate report messages must also do this.

² A *MsgId* generated by the queue manager consists of a 4-byte product identifier (AMQ- or CSQ- in either ASCII or EBCDIC, where - represents a blank character), followed by a product-specific implementation of a unique string. In IBM MQ this contains the first 12 characters of the queue-manager name, and a value derived from the system clock. All queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, in order to ensure that message identifiers are unique. The ability to generate a unique string also depends on the system clock not being changed backward. To eliminate the possibility of a message identifier generated by the queue manager duplicating one generated by the application, the application must avoid generating identifiers with initial characters in the range A through I in ASCII or EBCDIC (X'41' through X'49' and X'C1' through X'C9'). However, the application is not prevented from generating identifiers with initial characters in these ranges.

For the MQGET call, *MsgId* is one of the five fields that can be used to retrieve a particular message from the queue. Normally the MQGET call returns the next message on the queue, but a particular message can be obtained by specifying one or more of the five selection criteria, in any combination; these fields are:

- *MsgId*
- *CorrelId*
- *GroupId*
- *MsgSeqNumber*
- *Offset*

The application sets one or more of these field to the values required, and then sets the corresponding MQMO_* match options in the *MatchOptions* field in MQGMO to use those fields as selection criteria. Only messages that have the specified values in those fields are candidates for retrieval. The default for the *MatchOptions* field (if not altered by the application) is to match both the message identifier and the correlation identifier.

On z/OS, the selection criteria that you can use are restricted by the type of index used for the queue. See the *IndexType* queue attribute for further details.

Normally, the message returned is the *first* message on the queue that satisfies the selection criteria. But if MQGMO_BROWSE_NEXT is specified, the message returned is the *next* message that satisfies the selection criteria; the scan for this message starts with the message *following* the current cursor position.

Note: The queue is scanned sequentially for a message that satisfies the selection criteria, so retrieval times are slower than if no selection criteria are specified, especially if many messages have to be scanned before a suitable one is found. The exceptions to this are:

- an MQGET call by *CorrelId* on 64-bit distributed platforms where the *CorrelId* index eliminates the need to perform a true sequential scan.
- an MQGET call by *IndexType* on z/OS.

In both these cases, retrieval performance is improved.

See [Table 495 on page 369](#) for more information about how selection criteria are used in various situations.

Specifying MQMI_NONE as the message identifier has the same effect as *not* specifying MQMO_MATCH_MSG_ID, that is, *any* message identifier matches.

This field is ignored if the MQGMO_MSG_UNDER_CURSOR option is specified in the *GetMsgOpts* parameter on the MQGET call.

On return from an MQGET call, the *MsgId* field is set to the message identifier of the message returned (if any).

The following special value can be used:

MQMI_NONE

No message identifier is specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQMI_NONE_ARRAY is also defined; this has the same value as MQMI_NONE, but is an array of characters instead of a string.

This is an input/output field for the MQGET, MQPUT, and MQPUT1 calls. The length of this field is given by MQ_MSG_ID_LENGTH. The initial value of this field is MQMI_NONE.

MsgSeqNumber (MQLONG)

This is the sequence number of a logical message within a group.

Sequence numbers start at 1, and increase by 1 for each new logical message in the group, up to a maximum of 999 999 999. A physical message that is not in a group has a sequence number of 1.

The application does not have to set this field on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_MSG_SEQ_NUMBER is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *MsgSeqNumber* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value described in [Physical order on a queue](#). On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

On input to the MQGET call, the queue manager uses the value shown in [Table 495 on page 369](#). On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is one. This field is ignored if *Version* is less than MQMD_VERSION_2.

MsgType (MQLONG)

This indicates the type of the message. Message types are grouped as follows:

MQMT_SYSTEM_FIRST

Lowest value for system-defined message types.

MQMT_SYSTEM_LAST

Highest value for system-defined message types.

The following values are currently defined within the system range:

MQMT_DATAGRAM

The message is one that does not require a reply.

MQMT_REQUEST

The message is one that requires a reply.

Specify the name of the queue to which to send the reply in the *ReplyToQ* field. The *Report* field indicates how to set the *MsgId* and *CorrelId* of the reply.

MQMT_REPLY

The message is the reply to an earlier request message (MQMT_REQUEST). The message must be sent to the queue indicated by the *ReplyToQ* field of the request message. Use the *Report* field of the request to control how to set the *MsgId* and *CorrelId* of the reply.

Note: The queue manager does not enforce the request-reply relationship; this is an application responsibility.

MQMT_REPORT

The message is reporting on some expected or unexpected occurrence, usually related to some other message (for example, a request message was received that contained data that was not valid). Send the message to the queue indicated by the *ReplyToQ* field of the message descriptor of the original message. Set the *Feedback* field s to indicate the nature of the report. Use the *Report* field of the original message to control how to set the *MsgId* and *CorrelId* of the report message.

Report messages generated by the queue manager or message channel agent are always sent to the *ReplyToQ* queue, with the *Feedback* and *CorrelId* fields set as described above.

Application-defined values can also be used. They must be within the following range:

MQMT_APPL_FIRST

Lowest value for application-defined message types.

MQMT_APPL_LAST

Highest value for application-defined message types.

For the MQPUT and MQPUT1 calls, the *MsgType* value must be within either the system-defined range or the application-defined range; if it is not, the call fails with reason code MQRC_MSG_TYPE_ERROR.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is MQMT_DATAGRAM.

Offset (MQLONG)

This is the offset in bytes of the data in the physical message from the start of the logical message of which the data forms part. This data is called a *segment*. The offset is in the range 0 through 999 999 999. A physical message that is not a segment of a logical message has an offset of zero.

The application does not need to set this field on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_OFFSET is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application does not comply with these conditions, or the call is MQPUT1, the application must ensure that *Offset* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value described in [Physical order on a queue](#). On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

For a report message reporting on a segment of a logical message, the *OriginalLength* field (provided it is not MQOL_UNDEFINED) is used to update the offset in the segment information retained by the queue manager.

On input to the MQGET call, the queue manager uses the value shown in [Table 495 on page 369](#). On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is zero. This field is ignored if *Version* is less than MQMD_VERSION_2.

OriginalLength (MQLONG)

This field is relevant only for report messages that are segments. It specifies the length of the message segment to which the report message relates; it does not specify the length of the logical message of which the segment forms part, or the length of the data in the report message.

Note: When generating a report message for a message that is a segment, the queue manager and message channel agent copy into the MQMD for the report message the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags*, fields from the original message. As a result, the report message is also a segment. Applications that generate report messages must do the same, and set the *OriginalLength* field correctly.

The following special value is defined:

MQOL_UNDEFINED

Original length of message not defined.

OriginalLength is an input field on the MQPUT and MQPUT1 calls, but the value that the application provides is accepted only in particular circumstances:

- If the message being put is a segment and is also a report message, the queue manager accepts the value specified. The value must be:
 - Greater than zero if the segment is not the last segment
 - Not less than zero if the segment is the last segment
 - Not less than the length of data present in the message

If these conditions are not satisfied, the call fails with reason code MQRC_ORIGINAL_LENGTH_ERROR.

- If the message being put is a segment but not a report message, the queue manager ignores the field and uses the length of the application message data instead.
- In all other cases, the queue manager ignores the field and uses the value MQOL_UNDEFINED instead.

This is an output field on the MQGET call.

The initial value of this field is MQOL_UNDEFINED. This field is ignored if *Version* is less than MQMD_VERSION_2.

Persistence (MQLONG)

This indicates whether the message survives system failures and restarts of the queue manager. For the MQPUT and MQPUT1 calls, the value must be one of the following:

MQPER_PERSISTENT

The message survives system failures and restarts of the queue manager. Once the message has been put, and the unit of work in which it was put has been committed (if the message is put as part of a unit of work), the message is preserved on auxiliary storage. It remains there until the message is removed from the queue, and the unit of work in which it was got has been committed (if the message is retrieved as part of a unit of work).

When a persistent message is sent to a remote queue, a store-and-forward mechanism holds the message at each queue manager along the route to the destination, until the message is known to have arrived at the next queue manager.

Persistent messages cannot be placed on:

- Temporary dynamic queues
- Shared queues that map to a CFSTRUCT object at CFLEVEL(2) or below, or where the CFSTRUCT object is defined as RECOVER(NO).

Persistent messages can be placed on permanent dynamic queues, and predefined queues.

MQPER_NOT_PERSISTENT

The message does not usually survive system failures or queue manager restarts. This applies even if an intact copy of the message is found on auxiliary storage when the queue manager restarts.

In the case of NPMCLASS (HIGH) queues nonpersistent messages survive a normal queue manager shutdown and restart.

In the case of shared queues, nonpersistent messages survive queue manager restarts in the queue-sharing group, but do not survive failures of the coupling facility used to store messages on the shared queues.

MQPER_PERSISTENCE_AS_Q_DEF

- If the queue is a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed. Usually, all instances of a cluster queue have the same value for the *DefPersistence* attribute, although this is not mandated.

The value of *DefPersistence* is copied into the *Persistence* field when the message is placed on the destination queue. If *DefPersistence* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *local* queue manager, even if the destination queue manager is remote.

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path. This can be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPersistence* is copied into the *Persistence* field when the message is put. If *DefPersistence* is changed subsequently, messages that have already been put are not affected.

Both persistent and nonpersistent messages can exist on the same queue.

When replying to a message, applications must use the persistence of the request message for the reply message.

For an MQGET call, the value returned is either MQPER_PERSISTENT or MQPER_NOT_PERSISTENT.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQPER_PERSISTENCE_AS_Q_DEF.

Priority (MQLONG)

For the MQPUT and MQPUT1 calls, the value must be greater than or equal to zero; zero is the lowest priority. The following special value can also be used:

MQPRI_PRIORITY_AS_Q_DEF

- If the queue is a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed. Usually, all instances of a cluster queue have the same value for the *DefPriority* attribute, although this is not mandated.

The value of *DefPriority* is copied into the *Priority* field when the message is placed on the destination queue. If *DefPriority* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *local* queue manager, even if the destination queue manager is remote.

If there is more than one definition in the queue-name resolution path, the default priority is taken from the value of this attribute in the *first* definition in the path. This can be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPriority* is copied into the *Priority* field when the message is put. If *DefPriority* is changed subsequently, messages that have already been put are not affected.

The value returned by the MQGET call is always greater than or equal to zero; the value MQPRI_PRIORITY_AS_Q_DEF is never returned.

If a message is put with a priority greater than the maximum supported by the local queue manager (this maximum is given by the *MaxPriority* queue-manager attribute), the message is accepted by the queue manager, but placed on the queue at the queue manager's maximum priority; the MQPUT or MQPUT1 call completes with MQCC_WARNING and reason code MQRC_PRIORITY_EXCEEDS_MAXIMUM. However, the *Priority* field retains the value specified by the application that put the message.

On z/OS, if a message with a MsgSeqNumber of 1 is put to a queue that has a message delivery sequence of MQMDS_PRIORITY and an index type of MQIT_GROUP_ID, the queue might treat the message with a different priority. If the message was placed on the queue with a priority of 0 or 1, it is processed as though it has a priority of 2. This is because the order of messages placed on this type of queue is optimized to enable efficient group completeness tests. For more information on the message delivery sequence MQMDS_PRIORITY and the index type MQIT_GROUP_ID, see [MsgDeliverySequence attribute](#).

When replying to a message, applications must use the priority of the request message for the reply message. In other situations, specifying MQPRI_PRIORITY_AS_Q_DEF allows priority tuning to be carried out without changing the application.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQPRI_PRIORITY_AS_Q_DEF.


PutApplName (MQCHAR28)

This is the name of application that put the message, and is part of the *origin context* of the message. The contents differ between platforms, and might differ between releases.




For more information about message context, see [“Overview for MQMD” on page 397](#) and [Message context](#).

The format of *PutApplName* depends on the value of *PutApplType* and can change from one release to another. Changes are rare, but do happen if the environment changes.

When the queue manager sets this field (that is, for all options except MQPMO_SET_ALL_CONTEXT), it sets the field to a value that is determined by the environment:

-  **z/OS** On z/OS, the queue manager uses:
 - For z/OS batch, the 8-character job name from the JES JOB card
 - For TSO, the 7-character TSO user identifier
 - For CICS, the 8-character applid, followed by the 4-character tranid
 - For IMS, the 8-character IMS system identifier, followed by the 8-character PSB name
 - For XCF, the 8-character XCF group name, followed by the 16-character XCF member name
 - For a message generated by a queue manager, the first 28 characters of the queue manager name
 - For distributed queuing without CICS, the 8-character jobname of the channel initiator followed by the 8-character name of the module putting to the dead-letter queue followed by an 8-character task identifier.

The name or names are each padded to the right with blanks, as is any space in the remainder of the field. Where there is more than one name, there is no separator between them.

-  **Windows** On Windows systems, the queue manager uses the following names:
 - For a CICS application, the CICS transaction name
 - For a non-CICS application, the rightmost 28 characters of the fully-qualified name of the executable
-  **IBM i** On IBM i, the queue manager uses the fully-qualified job name.
-  **UNIX** On UNIX systems, the queue manager uses the following names:
 - For a CICS application, the CICS transaction name
 - For a non-CICS application, MQ asks the operating system for the name of the process. This is returned as the program file name, without full path. Then MQ places this process name in the MQMD.PutApplName field as follows:

AIX

If the name is less than or equal to 28 bytes, then the name is inserted, padded to the right with spaces.

If the name is greater than 28 bytes, then the leftmost 28 bytes of the name are inserted.

Linux and Solaris

If the name is less than or equal to 15 bytes, then the name is inserted, padded to the right with spaces.

If the name is greater than 15 bytes, then the leftmost 15 bytes of the name are inserted, padded to the right with spaces.

HP-UX

If the name is less than or equal to 14 bytes, then the name is inserted, padded to the right with spaces.

If the name is greater than 14 bytes, then the leftmost 14 bytes of the name are inserted, padded to the right with spaces.

For example, if you run `/opt/mqm/samp/bin/amqspuT QNAME QMNAME`, then the *PutApplName* is 'amqspuT '. There are 21 space characters of padding in

this MQCHAR28 field. Note that the full path including /opt/mqm/samp/bin is not included in the PutApplName.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

PutApplType (MQLONG)

This is the type of application that put the message, and is part of the **origin context** of the message. For more information about message context, see [“Overview for MQMD” on page 397](#) and [Message context](#).

PutApplType can have one of the following standard types. You can also define your own types, but only with values in the range MQAT_USER_FIRST through MQAT_USER_LAST.

MQAT_AIX

AIX application (same value as MQAT_UNIX).

▶ V 8.0.0.4 MQAT_AMQP

AMQP protocol application

MQAT_BROKER

Broker.

MQAT_CICS

CICS transaction.

MQAT_CICS_BRIDGE

CICS bridge.

MQAT_CICS_VSE

CICS/VSE transaction.

MQAT_DOS

IBM MQ MQI client application on PC DOS.

MQAT_DQM

Distributed queue manager agent.

MQAT_GUARDIAN

Tandem Guardian application (same value as MQAT_NSK).

MQAT_IMS

IMS application.

MQAT_IMS_BRIDGE

IMS bridge.

MQAT_JAVA

Java.

MQAT_MVS

MVS or TSO application (same value as MQAT_ZOS).

MQAT_NOTES_AGENT

Lotus Notes Agent application.

MQAT_NSK

HP Integrity NonStop Server application.

MQAT_OS390

OS/390® application (same value as MQAT_ZOS).

MQAT_OS400

IBM i application.

MQAT_QMGR

Queue manager.

MQAT_UNIX

UNIX application.

MQAT_VOS

Stratus VOS application.

MQAT_WINDOWS

16-bit Windows application.

MQAT_WINDOWS_NT

32-bit Windows application.

MQAT_WLM

z/OS workload manager application.

MQAT_XCF

XCF.

MQAT_ZOS

z/OS application.

MQAT_DEFAULT

Default application type.

This is the default application type for the platform on which the application is running.

Note: The value of this constant is environment-specific. Because of this, always compile the application using the header, include, or COPY files that are appropriate to the platform on which the application will run.

MQAT_UNKNOWN

Use this value to indicate that the application type is unknown, even though other context information is present.

MQAT_USER_FIRST

Lowest value for user-defined application type.

MQAT_USER_LAST

Highest value for user-defined application type.

The following special value can also occur:

MQAT_NO_CONTEXT

This value is set by the queue manager when a message is put with no context (that is, the MQPMO_NO_CONTEXT context option is specified).

When a message is retrieved, *PutApplType* can be tested for this value to decide whether the message has context (it is recommended that *PutApplType* is never set to MQAT_NO_CONTEXT, by an application using MQPMO_SET_ALL_CONTEXT, if any of the other context fields are nonblank).

When the queue manager generates this information as a result of an application put, the field is set to a value that is determined by the environment. On IBM i, it is set to MQAT_OS400; the queue manager never uses MQAT_CICS on IBM i.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

This is an output field for the MQGET call. The initial value of this field is MQAT_NO_CONTEXT.

PutDate (MQCHAR8)

This is the date when the message was put, and is part of the **origin context** of the message. For more information about message context, see [“Overview for MQMD” on page 397](#) and [Message context](#).

The format used for the date when this field is generated by the queue manager is:

- YYYYMMDD

where the characters represent:

YYYY

year (four numeric digits)

MM

month of year (01 through 12)

DD

day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

If the message was put as part of a unit of work, the date is that when the message was put, and not the date when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. The contents of the field are not checked by the queue manager, except that any information following a null character within the field is discarded. The queue manager converts the null character and any following characters to blanks. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

This is an output field for the MQGET call. The length of this field is given by MQ_PUT_DATE_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

PutTime (MQCHAR8)

This is the time when the message was put, and is part of the **origin context** of the message. For more information about message context, see [“Overview for MQMD” on page 397](#) and [Message context](#).

The format used for the time when this field is generated by the queue manager is:

- HHMMSSSTH

where the characters represent (in order):

HH

hours (00 through 23)

MM

minutes (00 through 59)

SS

seconds (00 through 59; see note)

T

tenths of a second (0 through 9)

H

hundredths of a second (0 through 9)

Note: If the system clock is synchronized to a very accurate time standard, it is possible on rare occasions for 60 or 61 to be returned for the seconds in *PutTime*. This happens when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

If the message was put as part of a unit of work, the time is that when the message was put, and not the time when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. The queue manager does not check the contents of the field, except that any information following a null character within the field is discarded. The queue manager converts the null character and any following characters to blanks. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

This is an output field for the MQGET call. The length of this field is given by MQ_PUT_TIME_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

ReplyToQ (MQCHAR48)

This is the name of the message queue to which the application that issued the get request for the message sends MQMT_REPLY and MQMT_REPORT messages. The name is the local name of a queue that is defined on the queue manager identified by *ReplyToQMgr*. This queue must not be a model queue, although the sending queue manager does not verify this when the message is put.

For the MQPUT and MQPUT1 calls, this field must not be blank if the *MsgType* field has the value MQMT_REQUEST, or if any report messages are requested by the *Report* field. However, the value specified (or substituted) is passed on to the application that issues the get request for the message, whatever the message type.

If the *ReplyToQMgr* field is blank, the local queue manager looks up the *ReplyToQ* name in its own queue definitions. If a local definition of a remote queue exists with this name, the *ReplyToQ* value in the transmitted message is replaced by the value of the *RemoteQName* attribute from the definition of the remote queue, and this value is returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, *ReplyToQ* is unchanged.

If the name is specified, it can contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise no check is made that the name satisfies the naming rules for queues; this is also true for the name transmitted, if the *ReplyToQ* is replaced in the transmitted message. The only check made is that a name has been specified, if the circumstances require it.

If a reply-to queue is not required, set the *ReplyToQ* field to blanks, or (in the C programming language) to the null string, or to one or more blanks followed by a null character; do not leave the field uninitialized.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

If a message that requires a report message cannot be delivered, and the report message also cannot be delivered to the queue specified, both the original message and the report message go to the dead-letter (undelivered-message) queue (see the *DeadLetterQName* attribute described in [“Attributes for the queue manager”](#) on page 768).

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ReplyToQMgr (MQCHAR48)

This is the name of the queue manager to which to send the reply message or report message. *ReplyToQ* is the local name of a queue that is defined on this queue manager.

If the *ReplyToQMgr* field is blank, the local queue manager looks up the *ReplyToQ* name in its queue definitions. If a local definition of a remote queue exists with this name, the *ReplyToQMgr* value in the transmitted message is replaced by the value of the *RemoteQMgrName* attribute from the definition of the remote queue, and this value is returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, the *ReplyToQMgr* that is transmitted with the message is the name of the local queue manager.

If the name is specified, it can contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise no check is made that the name satisfies the naming rules for queue managers, or that this name is known to the sending queue manager; this is also true for the name transmitted, if the *ReplyToQMgr* is replaced in the transmitted message.

If a reply-to queue is not required, set the *ReplyToQMgr* field to blanks, or (in the C programming language) to the null string, or to one or more blanks followed by a null character; do not leave the field uninitialized.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

Report (MQLONG)

A report message is a message about another message, used to inform an application about expected or unexpected events that relate to the original message. The *Report* field enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also (for both reports and replies) how the message and correlation identifiers in the report or reply message are to be set. Any or all (or none) of the following types of report message can be requested:

- Exception
- Expiration
- Confirm on arrival (COA)
- Confirm on delivery (COD)
- Positive action notification (PAN)
- Negative action notification (NAN)

You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

The application that receives the report message can determine the reason that the report was generated by examining the *Feedback* field in the MQMD; see the *Feedback* field for more details.

The use of report options when putting a message to a topic can cause zero, one, or many report messages to be generated and sent to the application. This is because the publication message may be sent to zero, one, or many subscribing applications.

Exception options: Specify one of the options listed to request an exception report message.

MQRO_EXCEPTION

A message channel agent generates this type of report when a message is sent to another queue manager and the message cannot be delivered to the specified destination queue. For example, the destination queue or an intermediate transmission queue might be full, or the message might be too big for the queue.

Generation of the exception report message depends on the persistence of the original message, and the speed of the message channel (normal or fast) through which the original message travels:

- For all persistent messages, and for nonpersistent messages traveling through normal message channels, the exception report is generated *only* if the action specified by the sending application for the error condition can be completed successfully. The sending application can specify one of the following actions to control the disposition of the original message when the error condition arises:
 - MQRO_DEAD_LETTER_Q (this places the original message on the dead-letter queue).
 - MQRO_DISCARD_MSG (this discards the original message).

If the action specified by the sending application cannot be completed successfully, the original message is left on the transmission queue, and no exception report message is generated.

- For nonpersistent messages traveling through fast message channels, the original message is removed from the transmission queue and the exception report generated *even if* the specified action for the error condition cannot be completed successfully. For example, if MQRO_DEAD_LETTER_Q is specified, but the original message cannot be placed on the dead-letter queue because that queue is full, the exception report message is generated and the original message discarded.

For more information about normal and fast message channels, see [Nonpersistent message speed \(NPMSPEED\)](#).

An exception report is not generated if the application that put the original message can be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call.

Applications can also send exception reports, to indicate that a message cannot be processed (for example, because it is a debit transaction that would cause the account to exceed its credit limit).

Message data from the original message is not included with the report message.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

MQRO_EXCEPTION_WITH_DATA

This is the same as MQRO_EXCEPTION, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

MQRO_EXCEPTION_WITH_FULL_DATA

Exception reports with full data required.

This is the same as MQRO_EXCEPTION, except that all the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

Expiration options: Specify one of the options listed to request an expiration report message.

MQRO_EXPIRATION

This type of report is generated by the queue manager if the message is discarded before delivery to an application because its expiry time has passed (see the *Expiry* field). If this option is not set, no report message is generated if a message is discarded for this reason (even if you specify one of the MQRO_EXCEPTION_* options).

Message data from the original message is not included with the report message.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

MQRO_EXPIRATION_WITH_DATA

This is the same as MQRO_EXPIRATION, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

MQRO_EXPIRATION_WITH_FULL_DATA

This is the same as MQRO_EXPIRATION, except that all the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

Confirm-on-arrival options: Specify one of the options listed to request a confirm-on-arrival report message.

MQRO_COA

This type of report is generated by the queue manager that owns the destination queue when the message is placed on the destination queue. Message data from the original message is not included with the report message.

If the message is put as part of a unit of work, and the destination queue is a local queue, the COA report message generated by the queue manager can be retrieved only if the unit of work is committed.

A COA report is not generated if the *Format* field in the message descriptor is MQFMT_XMIT_Q_HEADER or MQFMT_DEAD_LETTER_HEADER. This prevents a COA report being generated if the message is put on a transmission queue, or is undeliverable and put on a dead-letter queue.

In the case of an IMS bridge queue, the COA report is generated when the message reaches the IMS queue (acknowledgment received from IMS) and not when the message is put in the MQ bridge queue. That means that if IMS is not active, no COA report is generated until IMS is started and a message is queued on the IMS queue.

The user that runs a program that puts a message with MQMD.Report=MQRO_COA must have +passid authority on the reply queue. If the user does not have +passid authority, the COA report message does not reach the reply queue. An attempt is made to put the report message on the dead letter queue.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

MQRO_COA_WITH_DATA

This is the same as MQRO_COA, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

MQRO_COA_WITH_FULL_DATA

This is the same as MQRO_COA, except that all the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

Confirm-on-delivery options: Specify one of the options listed to request a confirm-on-delivery report message.

MQRO_COD

This type of report is generated by the queue manager when an application retrieves the message from the destination queue in a way that deletes the message from the queue. Message data from the original message is not included with the report message.

If the message is retrieved as part of a unit of work, the report message is generated within the same unit of work, so that the report is not available until the unit of work is committed. If the unit of work is backed out, the report is not sent.

A COD report is not always generated if a message is retrieved with the MQGMO_MARK_SKIP_BACKOUT option. If the primary unit of work is backed out but the secondary unit of work is committed, the message is removed from the queue, but a COD report is not generated.

A COD report is not generated if the *Format* field in the message descriptor is MQFMT_DEAD_LETTER_HEADER. This prevents a COD report being generated if the message is undeliverable and put on a dead-letter queue.

MQRO_COD is not valid if the destination queue is an XCF queue.

Do not specify more than one of MQRO_COD, MQRO_COD_WITH_DATA, and MQRO_COD_WITH_FULL_DATA.

MQRO_COD_WITH_DATA

This is the same as MQRO_COD, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or

more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

If `MQGMO_ACCEPT_TRUNCATED_MSG` is specified on the `MQGET` call for the original message, and the message retrieved is truncated, the amount of application message data placed in the report message depends on the environment:

- On z/OS, it is the minimum of:
 - The length of the original message
 - The length of the buffer used to retrieve the message
 - 100 bytes.
- In other environments, it is the minimum of:
 - The length of the original message
 - 100 bytes.

`MQRO_COD_WITH_DATA` is not valid if the destination queue is an XCF queue.

Do not specify more than one of `MQRO_COD`, `MQRO_COD_WITH_DATA`, and `MQRO_COD_WITH_FULL_DATA`.

MQRO_COD_WITH_FULL_DATA

This is the same as `MQRO_COD`, except that all the application message data from the original message is included in the report message.

`MQRO_COD_WITH_FULL_DATA` is not valid if the destination queue is an XCF queue.

Do not specify more than one of `MQRO_COD`, `MQRO_COD_WITH_DATA`, and `MQRO_COD_WITH_FULL_DATA`.

Action-notification options: Specify one or both of the options listed to request that the receiving application send a positive-action or negative-action report message.

MQRO_PAN

This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has been performed successfully. The application generating the report determines whether any data is to be included with the report.

Other than conveying this request to the application retrieving the message, the queue manager takes no action based on this option. The retrieving application must generate the report if appropriate.

MQRO_NAN

This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has *not* been performed successfully. The application generating the report determines whether any data is to be included with the report. For example, you might want to include some data indicating why the request could not be performed.

Other than conveying this request to the application retrieving the message, the queue manager takes no action based on this option. The retrieving application must generate the report if appropriate.

The application must determine which conditions correspond to a positive action and which correspond to a negative action. However, if the request has been only partially performed, generate a NAN report rather than a PAN report if requested. Every possible condition must correspond to either a positive action, or a negative action, but not both.

Message-identifier options: Specify one of the options listed to control how the *MsgId* of the report message (or of the reply message) is to be set.

MQRO_NEW_MSG_ID

This is the default action, and indicates that if a report or reply is generated as a result of this message, a new *MsgId* is generated for the report or reply message.

MQRO_PASS_MSG_ID

If a report or reply is generated as a result of this message, the *MsgId* of this message is copied to the *MsgId* of the report or reply message.

The *MsgId* of a publication message will be different for each subscriber that receives a copy of the publication and therefore the *MsgId* copied into the report or reply message will be different for each one.

If this option is not specified, MQRO_NEW_MSG_ID is assumed.

Correlation-identifier options: Specify one of the options listed to control how the *CorrelId* of the report message (or of the reply message) is to be set.

MQRO_COPY_MSG_ID_TO_CORREL_ID

This is the default action, and indicates that if a report or reply is generated as a result of this message, the *MsgId* of this message is copied to the *CorrelId* of the report or reply message.

The *MsgId* of a publication message will be different for each subscriber that receives a copy of the publication and therefore the *MsgId* copied into the *CorrelId* of the report or reply message will be different for each one.

MQRO_PASS_CORREL_ID

If a report or reply is generated as a result of this message, the *CorrelId* of this message is copied to the *CorrelId* of the report or reply message.

The *CorrelId* of a publication message will be specific to a subscriber unless it uses the MQSO_SET_CORREL_ID option and sets the SubCorrelId field in the MQSD to MQCI_NONE. Therefore it is possible that the *CorrelId* copied into the *CorrelId* of the report or reply message will be different for each one.

If this option is not specified, MQRO_COPY_MSG_ID_TO_CORREL_ID is assumed.

Servers replying to requests or generating report messages must check whether the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options were set in the original message. If they were, the servers must take the action described for those options. If neither is set, the servers must take the corresponding default action.

Disposition options: Specify one of the options listed to control the disposition of the original message when it cannot be delivered to the destination queue. The application can set the disposition options independently of requesting exception reports.

MQRO_DEAD_LETTER_Q

This is the default action, and places the message on the dead-letter queue if the message cannot be delivered to the destination queue. This happens in the following situations:

- When the application that put the original message cannot be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call. An exception report message is generated, if one was requested by the sender.
- When the application that put the original message was putting to a topic

MQRO_DISCARD_MSG

This discards the message if it cannot be delivered to the destination queue. This happens in the following situations:

- When the application that put the original message cannot be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call. An exception report message is generated, if one was requested by the sender.
- When the application that put the original message was putting to a topic

If you want to return the original message to the sender, without the original message being placed on the dead-letter queue, the sender must specify MQRO_DISCARD_MSG with MQRO_EXCEPTION_WITH_FULL_DATA.

MQRO_PASS_DISCARD_AND_EXPIRY

If this option is set on a message, and a report or reply is generated because of it, the message descriptor of the report inherits:

- MQRO_DISCARD_MSG if it was set.

- The remaining expiry time of the message (if this is not an expiry report). If this is an expiry report the expiry time is set to 60 seconds.

Activity option

MQRO_ACTIVITY

Using this value allows the route of **any** message to be traced throughout a queue manager network. The report option can be specified on any current user message, instantly allowing you to begin calculating the route of the message through the network.

If the application generating the message cannot switch on activity reports, reports can be turned on using an API crossing exit supplied by queue manager administrators.

Note:

1. The fewer the queue managers in the network that are able to generate activity reports, the less detailed the route.
2. The activity reports might be difficult to place in the correct order to determine the route taken.
3. The activity reports might not be able to find a route to their requested destination.
4. Messages with this report option set must be accepted by any queue manager, even if they do not understand the option. This allows the report option to be set on any user message, even if they are processed by a non Version 6.0 or later queue manager.
5. If a process, either a queue manager or a user process, performs an activity on a message with this option set it can choose to generate and put an activity report.

Default option: Specify the following if no report options are required:

MQRO_NONE

Use this value to indicate that no other options have been specified. MQRO_NONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

General information:

1. All report types required must be specifically requested by the application sending the original message. For example, if a COA report is requested but an exception report is not, a COA report is generated when the message is placed on the destination queue, but no exception report is generated if the destination queue is full when the message arrives there. If no *Report* options are set, no report messages are generated by the queue manager or message channel agent (MCA).

Some report options can be specified even though the local queue manager does not recognize them; this is useful when the option is to be processed by the *destination* queue manager. See [“Report options and message flags”](#) on page 866 for more details.

If a report message is requested, the name of the queue to which to send the report must be specified in the *ReplyToQ* field. When a report message is received, the nature of the report can be determined by examining the *Feedback* field in the message descriptor.

2. If the queue manager or MCA that generates a report message cannot put the report message on the reply queue (for example, because the reply queue or transmission queue is full), the report message is placed instead on the dead-letter queue. If that *also* fails, or there is no dead-letter queue, the action taken depends on the type of the report message:
 - If the report message is an exception report, the message that generated the exception report is left on its transmission queue; this ensures that the message is not lost.
 - For all other report types, the report message is discarded and processing continues normally. This is done because either the original message has already been delivered safely (for COA or COD report messages), or is no longer of any interest (for an expiration report message).

Once a report message has been placed successfully on a queue (either the destination queue or an intermediate transmission queue), the message is no longer subject to special processing; it is treated just like any other message.

3. When the report is generated, the *ReplyToQ* queue is opened and the report message put using the authority of the *UserIdentifier* in the MQMD of the message causing the report, except in the following cases:
 - Exception reports generated by a receiving MCA are put with whatever authority the MCA used when it tried to put the message causing the report.
 - COA reports generated by the queue manager are put with whatever authority was used when the message causing the report was put on the queue manager generating the report. For example, if the message was put by a receiving MCA using the MCA's user identifier, the queue manager puts the COA report using the MCA's user identifier.

Applications generating reports must use the same authority as they use to generate a reply; this is usually the authority of the user identifier in the original message.

If the report has to travel to a remote destination, senders and receivers can decide whether to accept it, in the same way as they do for other messages.

4. If a report message with data is requested:
 - The report message is always generated with the amount of data requested by the sender of the original message. If the report message is too big for the reply queue, the processing described above occurs; the report message is never truncated to fit on the reply queue.
 - If the *Format* of the original message is MQFMT_XMIT_Q_HEADER, the data included in the report does not include the MQXQH. The report data starts with the first byte of the data beyond the MQXQH in the original message. This occurs whether or not the queue is a transmission queue.
5. If a COA, COD, or expiration report message is received at the reply queue, it is guaranteed that the original message arrived, was delivered, or expired, as appropriate. However, if one or more of these report messages is requested and is *not* received, the reverse cannot be assumed, because one of the following might have occurred:
 - a. The report message is held up because a link is down.
 - b. The report message is held up because a blocking condition exists at an intermediate transmission queue or at the reply queue (for example, the queue is full or inhibited for puts).
 - c. The report message is on a dead-letter queue.
 - d. When the queue manager was attempting to generate the report message, it could neither put it on the appropriate queue, nor on the dead-letter queue, so the report message could not be generated.
 - e. A failure of the queue manager occurred between the action being reported (arrival, delivery, or expiry), and generation of the corresponding report message. (This does not happen for COD report messages if the application retrieves the original message within a unit of work, as the COD report message is generated within the same unit of work.)

Exception report messages can be held up in the same way for reasons 1, 2, and 3 above. However, when an MCA cannot generate an exception report message (the report message cannot be put either on the reply queue or the dead-letter queue), the original message remains on the transmission queue at the sender, and the channel is closed. This occurs irrespective of whether the report message was to be generated at the sending or the receiving end of the channel.

6. If the original message is temporarily blocked (resulting in an exception report message being generated and the original message being put on a dead-letter queue), but the blockage clears and an application then reads the original message from the dead-letter queue and puts it again to its destination, the following might occur:
 - Even though an exception report message has been generated, the original message eventually arrives successfully at its destination.
 - More than one exception report message is generated in respect of a single original message, because the original message might encounter another blockage later.

Report messages when putting to a topic:

1. Reports can be generated when putting a message to a topic. This message will be sent to all subscribers to the topic, which could be zero, one, or many. This should be taken into account when choosing to use report options as many report messages could be generated as a result.
2. When putting a message to a topic, there may be many destination queues that are to be given a copy of the message. If some of these destination queues have a problem, such as queue full, then the successful completion of the MQPUT depends on the setting of NPMMSGDLV or PMSGDLV (depending on the persistence of the message). If the setting is such that message delivery to the destination queue must be successful (for example, it is a persistent message to a durable subscriber and PMSGDLV is set to ALL or ALLDUR), then success is defined as one of the following criteria being met:
 - Successful put to the subscriber queue
 - Use of MQRO_DEAD_LETTER_Q and a successful put to the Dead-letter queue if the subscriber queue cannot take the message
 - Use of MQRO_DISCARD_MSG if the subscriber queue cannot take the message.

Report messages for message segments:

1. Report messages can be requested for messages that have segmentation allowed (see the description of the MQMF_SEGMENTATION_ALLOWED flag). If the queue manager finds it necessary to segment the message, a report message can be generated for each of the segments that subsequently encounters the relevant condition. Applications must be prepared to receive multiple report messages for each type of report message requested. Use the *GroupId* field in the report message to correlate the multiple reports with the group identifier of the original message, and the *Feedback* field identify the type of each report message.
2. If MQGMO_LOGICAL_ORDER is used to retrieve report messages for segments, be aware that reports of *different types* might be returned by the successive MQGET calls. For example, if both COA and COD reports are requested for a message that is segmented by the queue manager, the MQGET calls for the report messages might return the COA and COD report messages interleaved in an unpredictable fashion. Avoid this by using the MQGMO_COMPLETE_MSG option (optionally with MQGMO_ACCEPT_TRUNCATED_MSG). MQGMO_COMPLETE_MSG causes the queue manager to reassemble report messages that have the same report type. For example, the first MQGET call might reassemble all the COA messages relating to the original message, and the second MQGET call might reassemble all the COD messages. Which is reassembled first depends on which type of report message occurs first on the queue.
3. Applications that themselves put segments can specify different report options for each segment. However, note the following points:
 - If the segments are retrieved using the MQGMO_COMPLETE_MSG option, only the report options in the *first* segment are honored by the queue manager.
 - If the segments are retrieved one by one, and most of them have one of the MQRO_COD_* options, but at least one segment does not, you cannot use the MQGMO_COMPLETE_MSG option to retrieve the report messages with a single MQGET call, or use the MQGMO_ALL_SEGMENTS_AVAILABLE option to detect when all the report messages have arrived.
4. In an MQ network, the queue managers can have different capabilities. If a report message for a segment is generated by a queue manager or MCA that does not support segmentation, the queue manager or MCA does not by default include the necessary segment information in the report message, and this might make it difficult to identify the original message that caused the report to be generated. Avoid this difficulty by requesting data with the report message, that is, by specifying the appropriate MQRO_*_WITH_DATA or MQRO_*_WITH_FULL_DATA options. However, be aware that if MQRO_*_WITH_DATA is specified, *less than* 100 bytes of application message data might be returned to the application that retrieves the report message, if the report message is generated by a queue manager or MCA that does not support segmentation.

Contents of the message descriptor for a report message: When the queue manager or message channel agent (MCA) generates a report message, it sets the fields in the message descriptor to the following values, and then puts the message in the normal way.

Field in MQMD	Value used
<i>StrucId</i>	MQMD_STRUC_ID
<i>Version</i>	MQMD_VERSION_2
<i>Report</i>	MQRO_NONE
<i>MsgType</i>	MQMT_REPORT
<i>Expiry</i>	MQEI_UNLIMITED
<i>Feedback</i>	As appropriate for the nature of the report (MQFB_COA, MQFB_COD, MQFB_EXPIRATION, or an MQRC_* value)
<i>Encoding</i>	Copied from the original message descriptor
<i>CodedCharSetId</i>	Copied from the original message descriptor
<i>Format</i>	Copied from the original message descriptor
<i>Priority</i>	Copied from the original message descriptor
<i>Persistence</i>	Copied from the original message descriptor
<i>MsgId</i>	As specified by the report options in the original message descriptor
<i>CorrelId</i>	As specified by the report options in the original message descriptor
<i>BackoutCount</i>	0
<i>ReplyToQ</i>	Blanks
<i>ReplyToQMGr</i>	Name of queue manager
<i>UserIdentifier</i>	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
<i>AccountingToken</i>	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
<i>ApplIdentityData</i>	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
<i>PutApplType</i>	MQAT_QMGR, or as appropriate for the message channel agent
<i>PutApplName</i>	First 28 bytes of the queue-manager name or message channel agent name. For report messages generated by the IMS bridge, this field contains the XCF group name and XCF member name of the IMS system to which the message relates.
<i>PutDate</i>	Date when report message is sent
<i>PutTime</i>	Time when report message is sent
<i>ApplOriginData</i>	Blanks
<i>GroupId</i>	Copied from the original message descriptor
<i>MsgSeqNumber</i>	Copied from the original message descriptor
<i>Offset</i>	Copied from the original message descriptor
<i>MsgFlags</i>	Copied from the original message descriptor
<i>OriginalLength</i>	Copied from the original message descriptor if not MQOL_UNDEFINED, and set to the length of the original message data otherwise

An application generating a report is recommended to set similar values, except for the following:

- The *ReplyToQMGr* field can be set to blanks (the queue manager changes this to the name of the local queue manager when the message is put).
- Set the context fields using the option that would have been used for a reply, normally MQPMO_PASS_IDENTITY_CONTEXT.

Analyzing the report field: The *Report* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report must use one of the techniques described in [“Analyzing the report field”](#) on page 868.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQRO_NONE.

StrucId (MQCHAR4)

This is the structure identifier, and must be:

MQMD_STRUC_ID

Identifier for message descriptor structure.

For the C programming language, the constant MQMD_STRUC_ID_ARRAY is also defined; this has the same value as MQMD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQMD_STRUC_ID.

UserIdentifier (MQCHAR12)

This is part of the **identity context** of the message. For more information about message context, see [“Overview for MQMD”](#) on page 397 and [Message context](#).

UserIdentifier specifies the user identifier of the application that originated the message. The queue manager treats this information as character data, but does not define the format of it.

After a message has been received, use *UserIdentifier* in the *AlternateUserId* field of the *ObjDesc* parameter of a subsequent MQOPEN or MQPUT1 call to perform the authorization check for the *UserIdentifier* user instead of the application performing the open.

When the queue manager generates this information for an MQPUT or MQPUT1 call:

- On z/OS, the queue manager uses the *AlternateUserId* from the *ObjDesc* parameter of the MQOPEN or MQPUT1 call if the MQOO_ALTERNATE_USER_AUTHORITY or MQPMO_ALTERNATE_USER_AUTHORITY option was specified. If the relevant option was not specified, the queue manager uses a user identifier determined from the environment.
- In other environments, the queue manager always uses a user identifier determined from the environment.

When the user identifier is determined from the environment:

- On z/OS, the queue manager uses:
 - For MVS (batch), the user identifier from the JES JOB card or started task
 - For TSO, the user identifier propagated to the job during job submission
 - For CICS, the user identifier associated with the task
 - For IMS, the user identifier depends on the type of application:
 - For:
 - Nonmessage BMP regions
 - Nonmessage IFP regions
 - Message BMP and message IFP regions that have *not* issued a successful GU callthe queue manager uses the user identifier from the region JES JOB card or the TSO user identifier. If these are blank or null, it uses the name of the program specification block (PSB).
 - For:
 - Message BMP and message IFP regions that *have* issued a successful GU call
 - MPP regionsthe queue manager uses one of:
 - The signed-on user identifier associated with the message

- The logical terminal (LTERM) name
- The user identifier from the region JES JOB card
- The TSO user identifier
- The PSB name
- On IBM i, the queue manager uses the name of the user profile associated with the application job.
- On UNIX systems, the queue manager uses:
 - The application's logon name
 - The effective user identifier of the process if no logon is available
 - The user identifier associated with the transaction, if the application is a CICS transaction
- On Windows systems, the queue manager uses the first 12 characters of the logged-on user name.

This field is normally an output field generated by the queue manager but for an MQPUT or MQPUT1 call you can make this field an input/output field and specify the *UserIdentifier* field instead of letting the queue manager generate this information. Specify either MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT in the *PutMsgOpts* parameter and specify a user ID in the *UserIdentifier* field if you do not want the queue manager to generate the *UserIdentifier* field for an MQPUT or MQPUT1 call.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The queue manager converts the null character and any following characters to blanks. If MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *UserIdentifier* that was transmitted with the message if it was put to a queue. This will be the value of *UserIdentifier* that is kept with the message if it is retained (see description of MQPMO_RETAIN for more details about retained publications) but is not used as the *UserIdentifier* when the message is sent as a publication to subscribers because they provide a value to override *UserIdentifier* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_USER_ID_LENGTH. The initial value of this field is the null string in C, and 12 blank characters in other programming languages.

Version (MQLONG)

This is the structure version number, and must be one of the following:

MQMD_VERSION_1

Version-1 message descriptor structure.

This version is supported in all environments.

MQMD_VERSION_2

Version-2 message descriptor structure.

This version is supported in all IBM MQ V6.0 and later environments, plus IBM MQ MQI clients connected to these systems.

Note: When a version-2 MQMD is used, the queue manager performs additional checks on any MQ header structures that might be present at the beginning of the application message data; for further details see the usage notes for the MQPUT call.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQMD_CURRENT_VERSION

Current version of message descriptor structure.

This is always an input field. The initial value of this field is MQMD_VERSION_1.

Initial values and language declarations for MQMD

Table 504. Initial values of fields in MQMD for MQMD		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQMD_STRUC_ID	'MD'
<i>Version</i>	MQMD_VERSION_1	1
<i>Report</i>	MQRO_NONE	0
<i>MsgType</i>	MQMT_DATAGRAM	8
<i>Expiry</i>	MQEI_UNLIMITED	-1
<i>Feedback</i>	MQFB_NONE	0
<i>Encoding</i>	MQENC_NATIVE	Depends on environment
<i>CodedCharSetId</i>	MQCCSI_Q_MGR	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Priority</i>	MQPRI_PRIORITY_AS_Q_DEF	-1
<i>Persistence</i>	MQPER_PERSISTENCE_AS_Q_DEF	2
<i>MsgId</i>	MQMI_NONE	Nulls
<i>CorrelId</i>	MQCI_NONE	Nulls
<i>BackoutCount</i>	None	0
<i>ReplyToQ</i>	None	Null string or blanks
<i>ReplyToQMgr</i>	None	Null string or blanks
<i>UserIdentifier</i>	None	Null string or blanks
<i>AccountingToken</i>	MQACT_NONE	Nulls
<i>ApplIdentityData</i>	None	Null string or blanks
<i>PutApplType</i>	MQAT_NO_CONTEXT	0
<i>PutApplName</i>	None	Null string or blanks
<i>PutDate</i>	None	Null string or blanks
<i>PutTime</i>	None	Null string or blanks
<i>ApplOriginData</i>	None	Null string or blanks
<i>GroupId</i>	MQGI_NONE	Nulls
<i>MsgSeqNumber</i>	None	1
<i>Offset</i>	None	0
<i>MsgFlags</i>	MQMF_NONE	0
<i>OriginalLength</i>	MQOL_UNDEFINED	-1

Table 504. Initial values of fields in MQMD for MQMD (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 2. In the C programming language, the macro variable MQMD_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQMD MyMD = {MQMD_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQMD MQMD;
struct tagMQMD {
    MQCHAR4   StructId;           /* Structure identifier */
    MQLONG    Version;            /* Structure version number */
    MQLONG    Report;             /* Options for report messages */
    MQLONG    MsgType;            /* Message type */
    MQLONG    Expiry;             /* Message lifetime */
    MQLONG    Feedback;           /* Feedback or reason code */
    MQLONG    Encoding;           /* Numeric encoding of message data */
    MQLONG    CodedCharSetId;     /* Character set identifier of message
                                   data */
    MQCHAR8   Format;             /* Format name of message data */
    MQLONG    Priority;            /* Message priority */
    MQLONG    Persistence;        /* Message persistence */
    MQBYTE24  MsgId;              /* Message identifier */
    MQBYTE24  CorrelId;           /* Correlation identifier */
    MQLONG    BackoutCount;       /* Backout counter */
    MQCHAR48  ReplyToQ;           /* Name of reply queue */
    MQCHAR48  ReplyToQMGr;        /* Name of reply queue manager */
    MQCHAR12  UserIdentifier;      /* User identifier */
    MQBYTE32  AccountingToken;    /* Accounting token */
    MQCHAR32  ApplIdentityData;   /* Application data relating to
                                   identity */
    MQLONG    PutApplType;        /* Type of application that put the
                                   message */
    MQCHAR28  PutApplName;        /* Name of application that put the
                                   message */
    MQCHAR8   PutDate;            /* Date when message was put */
    MQCHAR8   PutTime;            /* Time when message was put */
    MQCHAR4   ApplOriginData;     /* Application data relating to origin */
    MQBYTE24  GroupId;            /* Group identifier */
    MQLONG    MsgSeqNumber;       /* Sequence number of logical message
                                   within group */
    MQLONG    Offset;             /* Offset of data in physical message
                                   from start of logical message */
    MQLONG    MsgFlags;           /* Message flags */
    MQLONG    OriginalLength;     /* Length of original message */
};
```

COBOL declaration

```
** MQMD structure
10 MQMD.
** Structure identifier
15 MQMD-STRUCID PIC X(4).
** Structure version number
15 MQMD-VERSION PIC S9(9) BINARY.
** Options for report messages
15 MQMD-REPORT PIC S9(9) BINARY.
** Message type
15 MQMD-MSGTYPE PIC S9(9) BINARY.
** Message lifetime
15 MQMD-EXPIRY PIC S9(9) BINARY.
** Feedback or reason code
15 MQMD-FEEDBACK PIC S9(9) BINARY.
** Numeric encoding of message data
15 MQMD-ENCODING PIC S9(9) BINARY.
```

```

** Character set identifier of message data
15 MQMD-CODEDCHARSETID PIC S9(9) BINARY.
** Format name of message data
15 MQMD-FORMAT PIC X(8).
** Message priority
15 MQMD-PRIORITY PIC S9(9) BINARY.
** Message persistence
15 MQMD-PERSISTENCE PIC S9(9) BINARY.
** Message identifier
15 MQMD-MSGID PIC X(24).
** Correlation identifier
15 MQMD-CORRELID PIC X(24).
** Backout counter
15 MQMD-BACKOUTCOUNT PIC S9(9) BINARY.
** Name of reply queue
15 MQMD-REPLYTOQ PIC X(48).
** Name of reply queue manager
15 MQMD-REPLYTOQMGR PIC X(48).
** User identifier
15 MQMD-USERIDENTIFIER PIC X(12).
** Accounting token
15 MQMD-ACCOUNTINGTOKEN PIC X(32).
** Application data relating to identity
15 MQMD-APPLIDENTITYDATA PIC X(32).
** Type of application that put the message
15 MQMD-PUTAPPLTYPE PIC S9(9) BINARY.
** Name of application that put the message
15 MQMD-PUTAPPLNAME PIC X(28).
** Date when message was put
15 MQMD-PUTDATE PIC X(8).
** Time when message was put
15 MQMD-PUTTIME PIC X(8).
** Application data relating to origin
15 MQMD-APPLORIGINDATA PIC X(4).
** Group identifier
15 MQMD-GROUPID PIC X(24).
** Sequence number of logical message within group
15 MQMD-MSGSEQUENBER PIC S9(9) BINARY.
** Offset of data in physical message from start of logical message
15 MQMD-OFFSET PIC S9(9) BINARY.
** Message flags
15 MQMD-MSGFLAGS PIC S9(9) BINARY.
** Length of original message
15 MQMD-ORIGINALLENGTH PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQMD based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version number */
    3 Report           fixed bin(31),    /* Options for report messages */
    3 MsgType          fixed bin(31),    /* Message type */
    3 Expiry           fixed bin(31),    /* Message lifetime */
    3 Feedback         fixed bin(31),    /* Feedback or reason code */
    3 Encoding         fixed bin(31),    /* Numeric encoding of message
                                     data */
    3 CodedCharSetId   fixed bin(31),    /* Character set identifier of
                                     message data */
    3 Format            char(8),          /* Format name of message data */
    3 Priority         fixed bin(31),    /* Message priority */
    3 Persistence      fixed bin(31),    /* Message persistence */
    3 MsgId           char(24),          /* Message identifier */
    3 CorrelId         char(24),          /* Correlation identifier */
    3 BackoutCount     fixed bin(31),    /* Backout counter */
    3 ReplyToQ         char(48),          /* Name of reply queue */
    3 ReplyToQMGR      char(48),          /* Name of reply queue manager */
    3 UserId           char(12),          /* User identifier */
    3 AccountingToken  char(32),          /* Accounting token */
    3 ApplIdentityData char(32),          /* Application data relating to
                                     identity */
    3 PutApplType      fixed bin(31),    /* Type of application that put the
                                     message */
    3 PutApplName      char(28),          /* Name of application that put the
                                     message */
    3 PutDate          char(8),          /* Date when message was put */
    3 PutTime          char(8),          /* Time when message was put */
    3 ApplOriginData   char(4),          /* Application data relating to
                                     origin */
    3 GroupId          char(24),          /* Group identifier */

```

3 MsgSeqNumber	fixed bin(31), /* Sequence number of logical message within group */
3 Offset	fixed bin(31), /* Offset of data in physical message from start of logical message */
3 MsgFlags	fixed bin(31), /* Message flags */
3 OriginalLength	fixed bin(31); /* Length of original message */

High Level Assembler declaration

MQMD	DSECT	
MQMD_STRUCID	DS	CL4 Structure identifier
MQMD_VERSION	DS	F Structure version number
MQMD_REPORT	DS	F Options for report messages
MQMD_MSGTYPE	DS	F Message type
MQMD_EXPIRY	DS	F Message lifetime
MQMD_FEEDBACK	DS	F Feedback or reason code
MQMD_ENCODING	DS	F Numeric encoding of message data
MQMD_CODEDCHARSETID	DS	F Character set identifier of message data
*		
MQMD_FORMAT	DS	CL8 Format name of message data
MQMD_PRIORITY	DS	F Message priority
MQMD_PERSISTENCE	DS	F Message persistence
MQMD_MSGID	DS	XL24 Message identifier
MQMD_CORRELID	DS	XL24 Correlation identifier
MQMD_BACKOUTCOUNT	DS	F Backout counter
MQMD_REPLYTOQ	DS	CL48 Name of reply queue
MQMD_REPLYTOQMGR	DS	CL48 Name of reply queue manager
MQMD_USERIDENTIFIER	DS	CL12 User identifier
MQMD_ACCOUNTINGTOKEN	DS	XL32 Accounting token
MQMD_APPLIDENTITYDATA	DS	CL32 Application data relating to identity
MQMD_PUTAPPLTYPE	DS	F Type of application that put the message
*		
MQMD_PUTAPPLNAME	DS	CL28 Name of application that put the message
*		
MQMD_PUTDATE	DS	CL8 Date when message was put
MQMD_PUTTIME	DS	CL8 Time when message was put
MQMD_APPLORIGINDATA	DS	CL4 Application data relating to origin
MQMD_GROUPID	DS	XL24 Group identifier
MQMD_MSGSEQNUMBER	DS	F Sequence number of logical message within group
*		
MQMD_OFFSET	DS	F Offset of data in physical message from start of logical message
*		
MQMD_MSGFLAGS	DS	F Message flags
MQMD_ORIGINALLENGTH	DS	F Length of original message
*		
MQMD_LENGTH	EQU	*-MQMD
	ORG	MQMD
MQMD_AREA	DS	CL(MQMD_LENGTH)

Visual Basic declaration

Type MQMD		
StrucId	As String*4	'Structure identifier'
Version	As Long	'Structure version number'
Report	As Long	'Options for report messages'
MsgType	As Long	'Message type'
Expiry	As Long	'Message lifetime'
Feedback	As Long	'Feedback or reason code'
Encoding	As Long	'Numeric encoding of message data'
CodedCharSetId	As Long	'Character set identifier of message data'
Format	As String*8	'Format name of message data'
Priority	As Long	'Message priority'
Persistence	As Long	'Message persistence'
MsgId	As MQBYTE24	'Message identifier'
CorrelId	As MQBYTE24	'Correlation identifier'
BackoutCount	As Long	'Backout counter'
ReplyToQ	As String*48	'Name of reply queue'
ReplyToQMgr	As String*48	'Name of reply queue manager'
UserIdentifier	As String*12	'User identifier'
AccountingToken	As MQBYTE32	'Accounting token'
ApplIdentityData	As String*32	'Application data relating to identity'
PutApplType	As Long	'Type of application that put the message'
PutApplName	As String*28	'Name of application that put the message'
PutDate	As String*8	'Date when message was put'

PutTime	As String*8	'Time when message was put'
ApplOriginData	As String*4	'Application data relating to origin'
GroupId	As MQBYTE24	'Group identifier'
MsgSeqNumber	As Long	'Sequence number of logical message' 'within group'
Offset	As Long	'Offset of data in physical message' 'from start of logical message'
MsgFlags	As Long	'Message flags'
OriginalLength	As Long	'Length of original message'
End Type		

MQMDE - Message descriptor extension

The following table summarizes the fields in the structure.

Table 505. Fields in MQMDE		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQMDE structure	StrucLength
<i>Encoding</i>	Numeric encoding of data that follows MQMDE	Encoding
<i>CodedCharSetId</i>	Character set identifier of data that follows MQMDE	CodedCharSetId
<i>Format</i>	Format name of data that follows MQMDE	Format
<i>Flags</i>	General flags	Flags
<i>GroupId</i>	Group identifier	GroupId
<i>MsgSeqNumber</i>	Sequence number of logical message within group	MsgSeqNumber
<i>Offset</i>	Offset of data in physical message from start of logical message	Offset
<i>MsgFlags</i>	Message flags	MsgFlags
<i>OriginalLength</i>	Length of original message	OriginalLength

Overview for MQMDE

Availability: All IBM MQ systems, plus IBM MQ clients connected to these systems.

Purpose: The MQMDE structure describes the data that sometimes occurs preceding the application message data. The structure contains those MQMD fields that exist in the version-2 MQMD, but not in the version-1 MQMD.

Format name: MQFMT_MD_EXTENSION.

Character set and encoding: Data in MQMDE must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE for the C programming language.

Set the character set and encoding of the MQMDE into the *CodedCharSetId* and *Encoding* fields in:

- The MQMD (if the MQMDE structure is at the start of the message data), or
- The header structure that precedes the MQMDE structure (all other cases).

If the MQMDE is not in the queue manager's character set and encoding, the MQMDE is accepted but not honored, that is, the MQMDE is treated as message data.

Note: On Windows, applications compiled with Micro Focus COBOL use a value of MQENC_NATIVE that is different from the queue-manager's encoding. Although numeric fields in the MQMD structure on the MQPUT, MQPUT1, and MQGET calls must be in the Micro Focus COBOL encoding, numeric fields in the MQMDE structure must be in the queue-manager's encoding. This latter is given by MQENC_NATIVE for the C programming language, and has the value 546.

Usage: Applications that use a version-2 MQMD will not encounter an MQMDE structure. However, specialized applications, and applications that continue to use a version-1 MQMD, might encounter an MQMDE in some situations. The MQMDE structure can occur in the following circumstances:

- Specified on the MQPUT and MQPUT1 calls
- Returned by the MQGET call
- In messages on transmission queues

MQMDE specified on MQPUT and MQPUT1 calls: On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *Format* field in MQMD to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager assumes default values for the fields in the MQMDE. The default values that the queue manager uses are the same as the initial values for the structure; see Table 507 on page 451.

If the application provides a version-2 MQMD *and* prefixes the application message data with an MQMDE, the structures are processed as shown in Table 506 on page 448.

Table 506. Queue-manager action when MQMDE specified on MQPUT or MQPUT1 for MQMDE			
MQMD version	Values of version-2 fields	Values of corresponding fields in MQMDE	Action taken by queue manager
1	-	Valid	MQMDE is honored
2	Default	Valid	MQMDE is honored
2	Not default	Valid	MQMDE is treated as message data
1 or 2	Any	Not valid	Call fails with an appropriate reason code
1 or 2	Any	MQMDE is in the wrong character set or encoding, or is an unsupported version	MQMDE is treated as message data
Note: On z/OS, if the application specifies a version-1 MQMD with an MQMDE, the queue manager validates the MQMDE only if the queue has an <i>IndexType</i> of MQIT_GROUP_ID.			

There is one special case. If the application uses a version-2 MQMD to put a message that is a segment (that is, the MQMF_SEGMENT or MQMF_LAST_SEGMENT flag is set), and the format name in the MQMD is MQFMT_DEAD_LETTER_HEADER, the queue manager generates an MQMDE structure and inserts it *between* the MQDLH structure and the data that follows it. In the MQMD that the queue manager retains with the message, the version-2 fields are set to their default values.

Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on MQPUT and MQPUT1. However, the queue manager does *not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

MQMDE returned by MQGET call: On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a nondefault value. The queue manager sets the *Format* field in MQMD to the value MQFMT_MD_EXTENSION to indicate that an MQMDE is present.

If the application provides an MQMDE at the start of the *Buffer* parameter, the MQMDE is ignored. On return from the MQGET call, it is replaced by the MQMDE for the message (if one is needed), or overwritten by the application message data (if the MQMDE is not needed).

If the MQGET call returns an MQMDE, the data in the MQMDE is usually in the queue manager's character set and encoding. However the MQMDE might be in some other character set and encoding if:

- The MQMDE was treated as data on the MQPUT or MQPUT1 call (see [Table 506 on page 448](#) for the circumstances that can cause this).
- The message was received from a remote queue manager connected by a TCP connection, and the receiving message channel agent (MCA) was not set up correctly.

Note: On Windows, applications compiled with Micro Focus COBOL use a value of MQENC_NATIVE that is different from the queue-manager's encoding (see above).

MQMDE in messages on transmission queues: Messages on transmission queues are prefixed with the MQXQH structure, which contains within it a version-1 MQMD. An MQMDE might also be present, positioned between the MQXQH structure and application message data, but it is usually present only if one or more of the fields in the MQMDE has a nondefault value.

Other MQ header structures can also occur between the MQXQH structure and the application message data. For example, when the dead-letter header MQDLH is present, and the message is not a segment, the order is:

- MQXQH (containing a version-1 MQMD)
- MQMDE
- MQDLH
- application message data

Fields for MQMDE

The MQMDE structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This specifies the character set identifier of the data that follows the MQMDE structure; it does not apply to character data in the MQMDE structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that this field is valid. The following special value can be used:

MQCCSI_INHERIT

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

MQCCSI_INHERIT cannot be used if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

This value is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

The initial value of this field is MQCCSI_UNDEFINED.

Encoding (MQLONG)

This specifies the numeric encoding of the data that follows the MQMDE structure; it does not apply to numeric data in the MQMDE structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that the field is valid. See the *Encoding* field described in [“MQMD - Message descriptor” on page 396](#) for more information about data encodings.

The initial value of this field is MQENC_NATIVE.

Flags (MQLONG)

The following flag can be specified:

MQMDEF_NONE

No flags.

The initial value of this field is MQMDEF_NONE.

Format (MQCHAR8)

This specifies the format name of the data that follows the MQMDE structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that this field is valid. See the *Format* field described in [“MQMD - Message descriptor”](#) on page 396 for more information about format names.

The initial value of this field is MQFMT_NONE.

GroupId (MQBYTE24)

See the *GroupId* field described in [“MQMD - Message descriptor”](#) on page 396. The initial value of this field is MQGI_NONE.

MsgFlags (MQLONG)

See the *MsgFlags* field described in [“MQMD - Message descriptor”](#) on page 396. The initial value of this field is MQMF_NONE.

MsgSeqNumber (MQLONG)

See the *MsgSeqNumber* field described in [“MQMD - Message descriptor”](#) on page 396. The initial value of this field is 1.

Offset (MQLONG)

See the *Offset* field described in [“MQMD - Message descriptor”](#) on page 396. The initial value of this field is 0.

OriginalLength (MQLONG)

See the *OriginalLength* field described in [“MQMD - Message descriptor”](#) on page 396. The initial value of this field is MQOL_UNDEFINED.

StrucId (MQCHAR4)

The value must be:

MQMDE_STRUC_ID

Identifier for message descriptor extension structure.

For the C programming language, the constant MQMDE_STRUC_ID_ARRAY is also defined; this has the same value as MQMDE_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQMDE_STRUC_ID.

StrucLength (MQLONG)

This is the length of the MQMDE structure; the following value is defined:

MQMDE_LENGTH_2

Length of version-2 message descriptor extension structure.

The initial value of this field is MQMDE_LENGTH_2.

Version (MQLONG)

This is the structure version number; the value must be:

MQMDE_VERSION_2

Version-2 message descriptor extension structure.

The following constant specifies the version number of the current version:

MQMDE_CURRENT_VERSION

Current version of message descriptor extension structure.

The initial value of this field is MQMDE_VERSION_2.

Initial values and language declarations for MQMDE

Table 507. Initial values of fields in MQMDE for MQMDE		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQMDE_STRUC_ID	'MDE¬'
<i>Version</i>	MQMDE_VERSION_2	2
<i>StrucLength</i>	MQMDE_LENGTH_2	72
<i>Encoding</i>	MQENC_NATIVE	Depends on environment
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQMDEF_NONE	0
<i>GroupId</i>	MQGI_NONE	Nulls
<i>MsgSeqNumber</i>	None	1
<i>Offset</i>	None	0
<i>MsgFlags</i>	MQMF_NONE	0
<i>OriginalLength</i>	MQOL_UNDEFINED	-1
Notes: 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQMDE_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQMDE MyMDE = {MQMDE_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQMDE MQMDE;
struct tagMQMDE {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     StrucLength;      /* Length of MQMDE structure */
    MQLONG     Encoding;         /* Numeric encoding of data that follows
                                MQMDE */
    MQLONG     CodedCharSetId;   /* Character-set identifier of data that
                                follows MQMDE */
    MQCHAR8    Format;           /* Format name of data that follows
                                MQMDE */
    MQLONG     Flags;            /* General flags */
    MQBYTE24   GroupId;          /* Group identifier */
    MQLONG     MsgSeqNumber;     /* Sequence number of logical message
                                within group */
    MQLONG     Offset;           /* Offset of data in physical message from
                                start of logical message */
    MQLONG     MsgFlags;         /* Message flags */
}
```

```

    MQLONG    OriginalLength; /* Length of original message */
};

```

COBOL declaration

```

**  MQMDE structure
10 MQMDE.
**  Structure identifier
15 MQMDE-STRUCID      PIC X(4).
**  Structure version number
15 MQMDE-VERSION      PIC S9(9) BINARY.
**  Length of MQMDE structure
15 MQMDE-STRUCLength PIC S9(9) BINARY.
**  Numeric encoding of data that follows MQMDE
15 MQMDE-ENCODING     PIC S9(9) BINARY.
**  Character-set identifier of data that follows MQMDE
15 MQMDE-CODEDCHARSETID PIC S9(9) BINARY.
**  Format name of data that follows MQMDE
15 MQMDE-FORMAT       PIC X(8).
**  General flags
15 MQMDE-FLAGS        PIC S9(9) BINARY.
**  Group identifier
15 MQMDE-GROUPID      PIC X(24).
**  Sequence number of logical message within group
15 MQMDE-MSGSEQNUMBER PIC S9(9) BINARY.
**  Offset of data in physical message from start of logical message
15 MQMDE-OFFSET       PIC S9(9) BINARY.
**  Message flags
15 MQMDE-MSGFLAGS     PIC S9(9) BINARY.
**  Length of original message
15 MQMDE-ORIGINALLENGTH PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQMDE based,
    3 StrucId      char(4),      /* Structure identifier */
    3 Version      fixed bin(31), /* Structure version number */
    3 StrucLength  fixed bin(31), /* Length of MQMDE structure */
    3 Encoding     fixed bin(31), /* Numeric encoding of data that
                                   follows MQMDE */
    3 CodedCharSetId fixed bin(31), /* Character-set identifier of data
                                   that follows MQMDE */
    3 Format        char(8),      /* Format name of data that follows
                                   MQMDE */
    3 Flags        fixed bin(31), /* General flags */
    3 GroupId      char(24),     /* Group identifier */
    3 MsgSeqNumber fixed bin(31), /* Sequence number of logical message
                                   within group */
    3 Offset       fixed bin(31), /* Offset of data in physical message
                                   from start of logical message */
    3 MsgFlags     fixed bin(31), /* Message flags */
    3 OriginalLength fixed bin(31); /* Length of original message */

```

High Level Assembler declaration

MQMDE	DSECT	
MQMDE_STRUCID	DS	CL4 Structure identifier
MQMDE_VERSION	DS	F Structure version number
MQMDE_STRUCLength	DS	F Length of MQMDE structure
MQMDE_ENCODING	DS	F Numeric encoding of data that follows MQMDE
* MQMDE_CODEDCHARSETID	DS	F Character-set identifier of data that follows MQMDE
* MQMDE_FORMAT	DS	CL8 Format name of data that follows MQMDE
MQMDE_FLAGS	DS	F General flags
MQMDE_GROUPID	DS	XL24 Group identifier
MQMDE_MSGSEQNUMBER	DS	F Sequence number of logical message within group
* MQMDE_OFFSET	DS	F Offset of data in physical message from start of logical message
* MQMDE_MSGFLAGS	DS	F Message flags
MQMDE_ORIGINALLENGTH	DS	F Length of original message
* MQMDE_LENGTH	EQU	*-MQMDE

MQMDE_AREA	ORG DS	MQMDE CL(MQMDE_LENGTH)
------------	--------	------------------------

Visual Basic declaration

```

Type MQMDE
    StrucId      As String*4 'Structure identifier'
    Version      As Long     'Structure version number'
    StrucLength  As Long     'Length of MQMDE structure'
    Encoding     As Long     'Numeric encoding of data that follows'
                                'MQMDE'
    CodedCharSetId As Long    'Character-set identifier of data that'
                                'follows MQMDE'
    Format       As String*8 'Format name of data that follows MQMDE'
    Flags        As Long     'General flags'
    GroupId      As MQBYTE24 'Group identifier'
    MsgSeqNumber As Long     'Sequence number of logical message within'
                                'group'
    Offset       As Long     'Offset of data in physical message from'
                                'start of logical message'
    MsgFlags     As Long     'Message flags'
    OriginalLength As Long   'Length of original message'
End Type

```

MQMHBO - Message handle to buffer options

The following table summarizes the fields in the structure. MQMHBO structure - message handle to buffer options

Table 508. Fields in MQMHBO		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options controlling the action of MQMHBUF	Options

Overview for MQMHBO

Availability: All IBM MQ systems and IBM MQ MQI clients.

Purpose: The MQMHBO structure allows applications to specify options that control how buffers are produced from message handles. The structure is an input parameter on the MQMHBUF call.

Character set and encoding: Data in MQMHBO must be in the character set of the application and encoding of the application (MQENC_NATIVE).

Fields for MQMHBO

Message handle to buffer options structure - fields

The MQMHBO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

Message handle to buffer options structure - Options field

These options control the action of MQMHBUF.

You must specify the following option:

MQMHBO_PROPERTIES_IN_MQRFH2

When converting properties from a message handle into a buffer, convert them into the MQRFH2 format.

Optionally, you can also specify the following option. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

MQMHBO_DELETE_PROPERTIES

Properties that are added to the buffer are deleted from the message handle. If the call fails no properties are deleted.

This is always an input field. The initial value of this field is MQMHBO_PROPERTIES_IN_MQRFH2.

StrucId (MQCHAR4)

Message handle to buffer options structure - StrucId field

This is the structure identifier. The value must be:

MQMHBO_STRUC_ID

Identifier for message handle to buffer options structure.

For the C programming language, the constant MQMHBO_STRUC_ID_ARRAY is also defined; this has the same value as MQMHBO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQMHBO_STRUC_ID.

Version (MQLONG)

Message handle to buffer options structure - Version field

This is the structure version number. The value must be:

MQMHBO_VERSION_1

Version number for message handle to buffer options structure.

The following constant specifies the version number of the current version:

MQMHBO_CURRENT_VERSION

Current version of message handle to buffer options structure.

This is always an input field. The initial value of this field is MQMHBO_VERSION_1.

Initial values and language declarations for MQMHBO

Message handle to buffer structure - Initial values

Table 509. Initial values of fields in MQMHBO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQMHBO_STRUC_ID	'MHBO '
<i>Version</i>	MQMHBO_VERSION_1	1
<i>Options</i>	MQMHBO_PROPERTIES_IN_MQRFH2	
Notes: 1. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 2. In the C programming language, the macro variable MQMHBO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQMHBO MyMHBO = {MQMHBO_DEFAULT};</pre>		

C declaration

Message handle to buffer options structure - C language declaration

```
typedef struct tagMQMHBO MQMHBO;  
struct tagMQMHBO {  
    MQCHAR4  StrucId;      /* Structure identifier */  
    MQLONG   Version;     /* Structure version number */  
    MQLONG   Options;     /* Options that control the action of
```

```

};
MQMHBUF */

```

COBOL declaration

Message handle to buffer options structure - COBOL language declaration

```

** MQMHBO structure
10 MQMHBO.
**   Structure identifier
15 MQMHBO-STRUCID          PIC X(4).
**   Structure version number
15 MQMHBO-VERSION         PIC S9(9) BINARY.
**   Options that control the action of MQMHBUF
15 MQMHBO-OPTIONS         PIC S9(9) BINARY.

```

PL/I declaration

Message handle to buffer options structure - PL/I language declaration

```

Dcl
1 MQMHBO based,
3 StrucId      char(4),      /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 Options      fixed bin(31), /* Options that control the action
                             of MQMHBUF */

```

High Level Assembler declaration

Message handle to buffer options structure - Assembler language declaration

```

MQMHBO          DSECT
MQMHBO_STRUCID  DS   CL4  Structure identifier
MQMHBO_VERSION DS   F    Structure version number
MQMHBO_OPTIONS DS   F    Options that control the
*               action of MQMHBUF
MQMHBO_LENGTH  EQU   *-MQMHBO
MQMHBO_AREA     DS   CL(MQMHBO_LENGTH)

```

MQOD - Object descriptor

The following table summarizes the fields in the structure.

Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>ObjectType</i>	Object type	ObjectType
<i>ObjectName</i>	Object name	ObjectName
<i>ObjectQMgrName</i>	Object queue manager name	ObjectQMgrName
<i>DynamicQName</i>	Dynamic queue name	DynamicQName
<i>AlternateUserId</i>	Alternate user identifier	AlternateUserId
Note: The remaining fields are ignored if <i>Version</i> is less than MQOD_VERSION_2.		
<i>RecsPresent</i>	Number of object records present	RecsPresent
<i>KnownDestCount</i>	Number of local queues opened successfully	KnownDestCount
<i>UnknownDestCount</i>	Number of remote queues opened successfully	UnknownDestCount
<i>InvalidDestCount</i>	Number of queues that failed to open	InvalidDestCount

Field	Description	Topic
<i>ObjectRecOffset</i>	Offset of first object record from start of MQOD	ObjectRecOffset
<i>ResponseRecOffset</i>	Offset of first response record from start of MQOD	ResponseRecOffset
<i>ObjectRecPtr</i>	Address of first object record	ObjectRecPtr
<i>ResponseRecPtr</i>	Address of first response record	ResponseRecPtr
Note: The remaining fields are ignored if <i>Version</i> is less than MQOD_VERSION_3.		
<i>AlternateSecurityId</i>	Alternate security identifier	AlternateSecurityId
<i>ResolvedQName</i>	Resolved queue name	ResolvedQName
<i>ResolvedQMgrName</i>	Resolved queue manager name	ResolvedQMgrName
Note: The remaining fields are ignored if <i>Version</i> is less than MQOD_VERSION_4.		
<i>ObjectString</i>	Long object name	ObjectString
<i>SelectionString</i>	Selection string	SelectionString
<i>ResObjectString</i>	Resolved long object name	ResObjectString
<i>ResolvedType</i>	Resolved object type	ResolvedType

Overview for MQOD

Availability: All IBM MQ systems, plus IBM MQ MQI clients connected to those systems.

Purpose: The MQOD structure is used to specify an object by name. The following types of object are valid:

- Queue or distribution list
- Namelist
- Process definition
- Queue manager
- Topic

The structure is an input/output parameter on the MQOPEN and MQPUT1 calls.

Version: The current version of MQOD is MQOD_VERSION_4. Applications that you want to port between several environments must ensure that the required version of MQOD is supported in all the environments concerned. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQOD that is supported by the environment, but with the initial value of the *Version* field set to MQOD_VERSION_1. To use fields that are not present in the version-1 structure, the application must set the *Version* field to the version number of the version required.

To open a distribution list, *Version* must be MQOD_VERSION_2 or greater.

Character set and encoding: Data in MQOD must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQOD

The MQOD structure contains the following fields; the fields are described in **alphabetical order**:

AlternateSecurityId (MQBYTE40)

This is a security identifier that is passed with the *AlternateUserId* to the authorization service to allow appropriate authorization checks to be performed. *AlternateSecurityId* is used only if:

- MQOO_ALTERNATE_USER_AUTHORITY is specified on the MQOPEN call, or
- MQPMO_ALTERNATE_USER_AUTHORITY is specified on the MQPUT1 call,

and the *AlternateUserId* field is not entirely blank up to the first null character or the end of the field.

On Windows, *AlternateSecurityId* can be used to supply the Windows security identifier (SID) that uniquely identifies the *AlternateUserId*. The SID for a user can be obtained from the Windows system by use of the `LookupAccountName()` Windows API call.

On z/OS, this field is ignored.

The *AlternateSecurityId* field has the following structure:

- The first byte is a binary integer containing the length of the significant data that follows; the value excludes the length byte itself. If no security identifier is present, the length is zero.
- The second byte indicates the type of security identifier that is present; the following values are possible:

MQSIDT_NT_SECURITY_ID

Windows security identifier.

MQSIDT_NONE

No security identifier.

- The third and subsequent bytes up to the length defined by the first byte contain the security identifier itself.
- Remaining bytes in the field are set to binary zero.

You can use the following special value:

MQSID_NONE

No security identifier specified.

The value is binary zero for the length of the field.

For the C programming language, the constant `MQSID_NONE_ARRAY` is also defined; this has the same value as `MQSID_NONE`, but is an array of characters instead of a string.

This is an input field. The length of this field is given by `MQ_SECURITY_ID_LENGTH`. The initial value of this field is `MQSID_NONE`. This field is ignored if *Version* is less than `MQOD_VERSION_3`.

AlternateUserId (MQCHAR12)

If you specify `MQOO_ALTERNATE_USER_AUTHORITY` for the MQOPEN call, or `MQPMO_ALTERNATE_USER_AUTHORITY` for the MQPUT1 call, this field contains an alternative user identifier that is used to check the authorization for the open, in place of the user identifier that the application is currently running under. Some checks, however, are still carried out with the current user identifier (for example, context checks).

If `MQOO_ALTERNATE_USER_AUTHORITY` or `MQPMO_ALTERNATE_USER_AUTHORITY` is specified and this field is entirely blank up to the first null character or the end of the field, the open can succeed only if no user authorization is needed to open this object with the options specified.

If neither `MQOO_ALTERNATE_USER_AUTHORITY` nor `MQPMO_ALTERNATE_USER_AUTHORITY` is specified, this field is ignored.

The following differences exist in the environments indicated:

- On z/OS, only the first 8 characters of *AlternateUserId* are used to check the authorization for the open. However, the current user identifier must be authorized to specify this particular alternative user identifier; all 12 characters of the alternative user identifier are used for this check. The user identifier must contain only characters allowed by the external security manager.

If *AlternateUserId* is specified for a queue, the value can be used subsequently by the queue manager when messages are put. If the MQPMO_*_CONTEXT options specified on the MQPUT or MQPUT1 call cause the queue manager to generate the identity context information, the queue manager places the *AlternateUserId* into the *UserIdentifier* field in the MQMD of the message, in place of the current user identifier.

- In other environments, *AlternateUserId* is used only for access control checks on the object being opened. If the object is a queue, *AlternateUserId* does not affect the content of the *UserIdentifier* field in the MQMD of messages sent using that queue handle.

This is an input field. The length of this field is given by MQ_USER_ID_LENGTH. The initial value of this field is the null string in C, and 12 blank characters in other programming languages.

DynamicQName (MQCHAR48)

This is the name of a dynamic queue that is to be created by the MQOPEN call. This is of relevance only when *ObjectName* specifies the name of a model queue; in all other cases *DynamicQName* is ignored.

The characters that are valid in the name are the same as those for *ObjectName*, except that an asterisk is also valid. A name that is blank (or one in which only blanks occur before the first null character) is not valid if *ObjectName* is the name of a model queue.

If the last nonblank character in the name is an asterisk (*), the queue manager replaces the asterisk with a string of characters that guarantees that the name generated for the queue is unique at the local queue manager. To allow a sufficient number of characters for this, the asterisk is valid only in positions 1 through 33. There must be no characters other than blanks or a null character following the asterisk.

It is valid for the asterisk to occur in the first character position, in which case the name consists solely of the characters generated by the queue manager.

On z/OS, do not use a name with the asterisk in the first character position, as there can be no security checks made on a queue with a full name that is generated automatically.

This is an input field. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is determined by the environment:

- On z/OS, the value is 'CSQ.*'.
- On other platforms, the value is 'AMQ.*'.

The value is a null-terminated string in C, and a blank-padded string in other programming languages.

InvalidDestCount (MQLONG)

This is the number of queues in the distribution list that failed to open successfully. If present, this field is also set when opening a single queue that is not in a distribution list.

Note: If present, this field is set *only* if the *CompCode* parameter on the MQOPEN or MQPUT1 call is MQCC_OK or MQCC_WARNING; it is *not* set if the *CompCode* parameter is MQCC_FAILED.

This is an output field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_1.

KnownDestCount (MQLONG)

This is the number of queues in the distribution list that resolve to local queues and that were opened successfully. The count does not include queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). If present, this field is also set when opening a single queue that is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_1.

ObjectName (MQCHAR48)

This is the local name of the object as defined on the queue manager identified by *ObjectQMGrName*. The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but can contain trailing blanks. Use a null character to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply in the environments indicated:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- On z/OS:
 - Avoid names that begin or end with an underscore; they cannot be processed by the operations and control panels.
 - The percent character has a special meaning to RACF. If RACF is used as the external security manager, names must not contain the percent. If they do, those names are not included in any security checks when RACF generic profiles are used.
- On IBM i, names containing lowercase characters, forward slash, or percent, must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified for names that occur as fields in structures or as parameters on calls.

The full topic name can be built from two different fields: *ObjectName* and *ObjectString*. For details of how these two fields are used, see [“Using topic strings” on page 554](#).

The following points apply to the types of object indicated:

- If *ObjectName* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ObjectName* field the name of the queue created. A model queue can be specified only on the MQOPEN call; a model queue is not valid on the MQPUT1 call.
- If *ObjectName* is the name of an alias queue with TARGTYPE(TOPIC), a security check is first made on the named alias queue; this is normal when alias queues are used. When the security check completes successfully, the MQOPEN call will continue and will behave like an MQOPEN call on an MQOT_TOPIC; this includes making a security check against the administrative topic object.
- If *ObjectName* and *ObjectQMGrName* identify a shared queue owned by the queue-sharing group to which the local queue manager belongs, there must not also be a queue definition of the same name on the local queue manager. If there is such a definition (a local queue, alias queue, remote queue, or model queue), the call fails with reason code MQRC_OBJECT_NOT_UNIQUE.
- If the object being opened is a distribution list (that is, *RecsPresent* is present and greater than zero), *ObjectName* must be blank or the null string. If this condition is not satisfied, the call fails with reason code MQRC_OBJECT_NAME_ERROR.
- If *ObjectType* is MQOT_Q_MGR, special rules apply; in this case the name must be entirely blank up to the first null character or the end of the field.

This is an input/output field for the MQOPEN call when *ObjectName* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ObjectQMGrName (MQCHAR48)

This is the name of the queue manager on which the *ObjectName* object is defined. The characters that are valid in the name are the same as those for *ObjectName* (see [“ObjectName \(MQCHAR48\)” on page 459](#)). A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected (the local queue manager).

The following points apply to the types of object indicated:

- If *ObjectType* is MQOT_TOPIC, MQOT_NAMELIST, MQOT_PROCESS, or MQOT_Q_MGR, *ObjectQMgrName* must be blank or the name of the local queue manager.
- If *ObjectName* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ObjectQMgrName* field the name of the queue manager on which the queue is created; this is the name of the local queue manager. A model queue can be specified only on the MQOPEN call; a model queue is not valid on the MQPUT1 call.
- If *ObjectName* is the name of a cluster queue, and *ObjectQMgrName* is blank, the destination of messages sent using the queue handle returned by the MQOPEN call is chosen by the queue manager (or cluster workload exit, if one is installed) as follows:
 - If MQOO_BIND_ON_OPEN is specified, the queue manager selects a particular instance of the cluster queue while processing the MQOPEN call, and all messages put using this queue handle are sent to that instance.
 - If MQOO_BIND_NOT_FIXED is specified, the queue manager can choose a different instance of the destination queue (residing on a different queue manager in the cluster) for each successive MQPUT call that uses this queue handle.

If the application needs to send a message to a *specific* instance of a cluster queue (that is, a queue instance that resides on a particular queue manager in the cluster), the application must specify the name of that queue manager in the *ObjectQMgrName* field. This forces the local queue manager to send the message to the specified destination queue manager.

- If *ObjectName* is the name of a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *ObjectQMgrName* can be the name of the queue-sharing group, the name of the local queue manager, or blank; the message is placed on the same queue whichever of these values is specified.

Queue-sharing groups are supported only on z/OS.

- If *ObjectName* is the name of a shared queue that is owned by a remote queue-sharing group (that is, a queue-sharing group to which the local queue manager does *not* belong), *ObjectQMgrName* must be the name of the queue-sharing group. You can use the name of a queue manager that belongs to that group, but this can delay the message if that particular queue manager is not available when the message arrives at the queue-sharing group.
- If the object being opened is a distribution list (that is, *RecsPresent* is greater than zero), *ObjectQMgrName* must be blank or the null string. If this condition is not satisfied, the call fails with reason code MQRC_OBJECT_Q_MGR_NAME_ERROR.

This is an input/output field for the MQOPEN call when *ObjectName* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ObjectRecOffset (MQLONG)

This is the offset in bytes of the first MQOR object record from the start of the MQOD structure. The offset can be positive or negative. *ObjectRecOffset* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

When a distribution list is being opened, an array of one or more MQOR object records must be provided in order to specify the names of the destination queues in the distribution list. This can be done in one of two ways:

- By using the offset field *ObjectRecOffset*.

In this case, the application must declare its own structure containing an MQOD followed by the array of MQOR records (with as many array elements as are needed), and set *ObjectRecOffset* to the offset of the first element in the array from the start of the MQOD. Ensure that this offset is correct and has a value that can be accommodated within an MQLONG (the most restrictive programming language is COBOL, for which the valid range is -999 999 999 through +999 999 999).

Use *ObjectRecOffset* for programming languages that do not support the pointer data type, or that implement the pointer data type in a way that is not portable to different environments (for example, the COBOL programming language).

- By using the pointer field *ObjectRecPtr*.

In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ObjectRecPtr* to the address of the array.

Use *ObjectRecPtr* for programming languages that support the pointer data type in a way that is portable to different environments (for example, the C programming language).

Whatever technique you choose, use one of *ObjectRecOffset* and *ObjectRecPtr*; the call fails with reason code MQRC_OBJECT_RECORDS_ERROR if both are zero, or both are nonzero.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_2.

ObjectRecPtr (MQPTR)

This is the address of the first MQOR object record. *ObjectRecPtr* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

You can use either *ObjectRecPtr* or *ObjectRecOffset* to specify the object records, but not both; for the description of the *ObjectRecOffset* field, see [“ObjectRecOffset \(MQLONG\)”](#) on page 460. If you do not use *ObjectRecPtr*, set it to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQOD_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

ObjectString (MQCHARV)

The *ObjectString* field specifies the long object name.

This specifies the long object name to be used. This field is only referenced for certain values of *ObjectType*, and is ignored for all other values. See the description of *ObjectType* for details of which values indicate that this field is used.

If *ObjectString* is specified incorrectly, according to the description of how to use the MQCHARV structure, or if it exceeds the maximum length, the call fails with reason code MQRC_OBJECT_STRING_ERROR.

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

The full topic name can be built from two different fields: *ObjectName* and *ObjectString*. For details of how these two fields are used, see [“Using topic strings”](#) on page 554.

ObjectType (MQLONG)

The type of object being named in the object descriptor. Possible values are:

MQOT_CLNTCONN_CHANNEL

Client connection channel. The name of the object is found in the *ObjectName* field.

MQOT_Q

Queue. The name of the object is found in the *ObjectName* field.

MQOT_NAMELIST

Namelist. The name of the object is found in the *ObjectName* field

MQOT_PROCESS

Process definition. The name of the object is found in the *ObjectName* field

MQOT_Q_MGR

Queue manager. The name of the object is found in the *ObjectName* field

MQOT_TOPIC

Topic. The full topic name can be built from two different fields: *ObjectName* and *ObjectString*.

For details of how those two fields are used, see [“Using topic strings” on page 554](#).

This is always an input field. The initial value of this field is MQOT_Q.

RecsPresent (MQLONG)

This is the number of MQOR object records that have been provided by the application. If this number is greater than zero, it indicates that a distribution list is being opened, with *RecsPresent* being the number of destination queues in the list. A distribution list can contain only one destination.

The value of *RecsPresent* must not be less than zero, and if it is greater than zero *ObjectType* must be MQOT_Q; the call fails with reason code MQRC_RECS_PRESENT_ERROR if these conditions are not satisfied.

On z/OS, this field must be zero.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_2.

ResObjectString (MQCHARV)

The *ResObjectString* field is the long object name after the queue manager resolves the name provided in the *ObjectName* field.

This field is returned only for topics and queue aliases that reference a topic object.

If the long object name is provided in *ObjectString* and nothing is provided in *ObjectName*, then the value returned in this field is the same as provided in *ObjectString*.

If this field is omitted (that is *ResObjectString.VSBufSize* is zero) then the *ResObjectString* will not be returned, but the length will be returned in *ResObjectString.VSLength*.

If the buffer length (provided in *ResObjectString.VSBufSize*) is shorter than the full *ResObjectString*, the string will be truncated and will return as many of the rightmost characters as can fit in the provided buffer.

If *ResObjectString* is specified incorrectly, according to the description of how to use the [MQCHARV](#) structure, or if it exceeds the maximum length, the call fails with reason code MQRC_RES_OBJECT_STRING_ERROR.

ResolvedQMgrName (MQCHAR48)

This is the name of the destination queue manager after the local queue manager resolves the name. The name returned is the name of the queue manager that owns the queue identified by *ResolvedQName*.

ResolvedQMgrName can be the name of the local queue manager.

If *ResolvedQName* is a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *ResolvedQMgrName* is the name of the queue-sharing group. If the queue is owned by some other queue-sharing group, *ResolvedQName* can be the name of the queue-sharing group or the name of a queue manager that is a member of the queue-sharing group (the nature of the value returned is determined by the queue definitions that exist at the local queue manager).

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *ResolvedQMgrName* is set to blanks:

- Not a queue
- A queue, but not opened for browse, input, or output
- A cluster queue with MQOO_BIND_NOT_FIXED specified (or with MQOO_BIND_AS_Q_DEF in effect when the *DefBind* queue attribute has the value MQBND_BIND_NOT_FIXED)
- A distribution list

This is an output field. The length of this field is given by `MQ_Q_NAME_LENGTH`. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is ignored if *Version* is less than `MQOD_VERSION_3`.

ResolvedQName (MQCHAR48)

This is the name of the destination queue after the local queue manager resolves the name. The name returned is the name of a queue that exists on the queue manager identified by *ResolvedQMGrName*.

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *ResolvedQName* is set to blanks:

- Not a queue
- A queue, but not opened for browse, input, or output
- A distribution list
- An alias queue that references a topic object (refer to *ResObjectString* instead).
- An alias queue that resolves to a topic object.

This is an output field. The length of this field is given by `MQ_Q_NAME_LENGTH`. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is ignored if *Version* is less than `MQOD_VERSION_3`.

ResolvedType (MQLONG)

The type of the resolved (base) object being opened.

The possible values are:

MQOT_Q

The resolved object is a queue. This value applies when a queue is opened directly or when an alias queue pointing to a queue is opened.

MQOT_TOPIC

The resolved object is a topic. This value applies when a topic is opened directly or when an alias queue pointing to a topic object is opened.

MQOT_NONE

The resolved type is neither a queue nor a topic.

ResponseRecOffset (MQLONG)

This is the offset in bytes of the first MQRR response record from the start of the MQOD structure. The offset can be positive or negative. *ResponseRecOffset* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

When a distribution list is being opened, you can provide an array of one or more MQRR response records in order to identify the queues that failed to open (*CompCode* field in MQRR), and the reason for each failure (*Reason* field in MQRR). The data is returned in the array of response records in the same order as the queue names occur in the array of object records. The queue manager sets the response records only when the outcome of the call is mixed (that is, some queues were opened successfully while others failed, or all failed but for different reasons); reason code `MQRC_MULTIPLE_REASONS` from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *Reason* parameter of the MQOPEN or MQPUT1 call, and the response records are not set. Response records are optional, but if they are supplied there must be *RecsPresent* of them.

The response records can be provided in the same way as the object records, either by specifying an offset in *ResponseRecOffset*, or by specifying an address in *ResponseRecPtr*; for details of how to do this, see [“ObjectRecOffset \(MQLONG\)” on page 460](#). However, no more than one of *ResponseRecOffset* and *ResponseRecPtr* can be used; the call fails with reason code `MQRC_RESPONSE_RECORDS_ERROR` if both are nonzero.

For the MQPUT1 call, these response records are used to return information about errors that occur when the message is sent to the queues in the distribution list, as well as errors that occur when the queues are opened. The completion code and reason code from the put operation for a queue replace those

from the open operation for that queue only if the completion code from the latter was MQCC_OK or MQCC_WARNING.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_2.

ResponseRecPtr (MQPTR)

This is the address of the first MQRR response record. *ResponseRecPtr* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

Use either *ResponseRecPtr* or *ResponseRecOffset* to specify the response records, but not both; for details, see “[ResponseRecOffset \(MQLONG\)](#)” on page 463. If you do not use *ResponseRecPtr*, set it to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQOD_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

SelectionString (MQCHARV)

This is the string used to provide the selection criteria used when retrieving messages off a queue.

SelectionString must not be provided in the following cases:

- If *ObjectType* is not MQOT_Q
- If the queue being opened is not being opened using one of the MQOO_BROWSE, or MQOO_INPUT_* options

If *SelectionString* is provided in these cases, the call fails with reason code MQRC_SELECTOR_INVALID_FOR_TYPE.

If *SelectionString* is specified incorrectly, according to the description of how to use the “[MQCHARV - Variable Length String](#)” on page 284 structure, or if it exceeds the maximum length, the call fails with reason code MQRC_SELECTION_STRING_ERROR. The maximum length of *SelectionString* is MQ_SELECTOR_LENGTH.

SelectionString usage is described in [Selectors](#).

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQOD_STRUC_ID

Identifier for object descriptor structure.

For the C programming language, the constant MQOD_STRUC_ID_ARRAY is also defined; this has the same value as MQOD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQOD_STRUC_ID.

UnknownDestCount (MQLONG)

This is the number of queues in the distribution list that resolve to remote queues and that were opened successfully. If present, this field is also set when opening a single queue that is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is ignored if *Version* is less than MQOD_VERSION_1.

Version (MQLONG)

This is the structure version number; the value must be one of the following:

MQOD_VERSION_1

Version-1 object descriptor structure.

MQOD_VERSION_2

Version-2 object descriptor structure.

MQOD_VERSION_3

Version-3 object descriptor structure.

MQOD_VERSION_4

Version-4 object descriptor structure.

All versions are supported in all IBM MQ V7.0 environments.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQOD_CURRENT_VERSION

Current version of object descriptor structure.

This is always an input field. The initial value of this field is MQOD_VERSION_1.

Initial values and language declarations for MQOD

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQOD_STRUC_ID	'OD- -'
<i>Version</i>	MQOD_VERSION_1	1
<i>ObjectType</i>	MQOT_Q	1
<i>ObjectName</i>	None	Null string or blanks
<i>ObjectQMgrName</i>	None	Null string or blanks
<i>DynamicQName</i>	None	'CSQ.*' on z/OS ; 'AMQ.*' otherwise
<i>AlternateUserId</i>	None	Null string or blanks
<i>RecsPresent</i>	None	0
<i>KnownDestCount</i>	None	0
<i>UnknownDestCount</i>	None	0
<i>InvalidDestCount</i>	None	0
<i>ObjectRecOffset</i>	None	0
<i>ResponseRecOffset</i>	None	0
<i>ObjectRecPtr</i>	None	Null pointer or null bytes
<i>ResponseRecPtr</i>	None	Null pointer or null bytes
<i>AlternateSecurityId</i>	MQSID_NONE	Nulls
<i>ResolvedQName</i>	None	Null string or blanks
<i>ResolvedQMgrName</i>	None	Null string or blanks
<i>ObjectString</i>	MQCHARV_DEFAULT	As defined for MQCHARV
<i>SelectionString</i>	MQCHARV_DEFAULT	As defined for MQCHARV
<i>ResObjectString</i>	MQCHARV_DEFAULT	As defined for MQCHARV
<i>ResolvedType</i>	MQOT_NONE	0

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQOD_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQOD MyOD = {MQOD_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQOD MQOD;
struct tagMQOD {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     ObjectType;       /* Object type */
    MQCHAR48   ObjectName;       /* Object name */
    MQCHAR48   ObjectQMgrName;   /* Object queue manager name */
    MQCHAR48   DynamicQName;     /* Dynamic queue name */
    MQCHAR12   AlternateUserId;  /* Alternate user identifier */
    /* Ver:1 */
    MQLONG     RecsPresent;       /* Number of object records present */
    MQLONG     KnownDestCount;    /* Number of local queues opened
                                   successfully */
    MQLONG     UnknownDestCount; /* Number of remote queues opened
                                   successfully */
    MQLONG     InvalidDestCount; /* Number of queues that failed to
                                   open */
    MQLONG     ObjectRecOffset;   /* Offset of first object record from
                                   start of MQOD */
    MQLONG     ResponseRecOffset; /* Offset of first response record
                                   from start of MQOD */
    MQPTR      ObjectRecPtr;      /* Address of first object record */
    MQPTR      ResponseRecPtr;    /* Address of first response record */
    /* Ver:2 */
    MQBYTE40   AlternateSecurityId; /* Alternate security identifier */
    MQCHAR48   ResolvedQName;      /* Resolved queue name */
    MQCHAR48   ResolvedQMgrName;   /* Resolved queue manager name */
    /* Ver:3 */
    MQCHARV    ObjectString;       /* Object Long name */
    MQCHARV    SelectionString;    /* Message Selector */
    MQCHARV    ResObjectString;    /* Resolved Long object name*/
    MQLONG     ResolvedType        /* Alias queue resolved
                                   object type */
    /* Ver:4 */
};
```

COBOL declaration

```
** MQOD structure
10  MQOD.
** Structure identifier
15  MQOD-STRUCID                PIC X(4).
** Structure version number
15  MQOD-VERSION                PIC S9(9) BINARY.
** Object type
15  MQOD-OBJECTTYPE            PIC S9(9) BINARY.
** Object name
15  MQOD-OBJECTNAME            PIC X(48).
** Object queue manager name
15  MQOD-OBJECTQMGRNAME        PIC X(48).
** Dynamic queue name
15  MQOD-DYNAMICQNAME          PIC X(48).
** Alternate user identifier
15  MQOD-ALTERNATEUSERID        PIC X(12).
** Number of object records present
15  MQOD-RECSPRESENT            PIC S9(9) BINARY.
** Number of local queues opened successfully
15  MQOD-KNOWNDESTCOUNT        PIC S9(9) BINARY.
```

```

** Number of remote queues opened successfully
15 MQOD-UNKNOWNDESTCOUNT PIC S9(9) BINARY.
** Number of queues that failed to open
15 MQOD-INVALIDDESTCOUNT PIC S9(9) BINARY.
** Offset of first object record from start of MQOD
15 MQOD-OBJECTRECOFFSET PIC S9(9) BINARY.
** Offset of first response record from start of MQOD
15 MQOD-RESPONSERECOFFSET PIC S9(9) BINARY.
** Address of first object record
15 MQOD-OBJECTRECPtr POINTER.
** Address of first response record
15 MQOD-RESPONSERECPtr POINTER.
** Alternate security identifier
15 MQOD-ALTERNATESECURITYID PIC X(40).
** Resolved queue name
15 MQOD-RESOLVEDQNAME PIC X(48).
** Resolved queue manager name
15 MQOD-RESOLVEDQMGRNAME PIC X(48).
** Object Long name
15 MQOD-OBJECTSTRING.
** Address of variable length string
20 MQOD-OBJECTSTRING-VSPTR POINTER.
** Offset of variable length string
20 MQOD-OBJECTSTRING-VSOFFSET PIC S9(9) BINARY.
** size of buffer
20 MQOD-OBJECTSTRING-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQOD-OBJECTSTRING-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQOD-OBJECTSTRING-VSCCSID PIC S9(9) BINARY.
** Message Selector
15 MQOD-SELECTIONSTRING.
** Address of variable length string
20 MQOD-SELECTIONSTRING-VSPTR POINTER.
** Offset of variable length string
20 MQOD-SELECTIONSTRING-VSOFFSET PIC S9(9) BINARY.
** size of buffer
20 MQOD-SELECTIONSTRING-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQOD-SELECTIONSTRING-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQOD-SELECTIONSTRING-VSCCSID PIC S9(9) BINARY.
** Resolved Long object name
15 MQOD-RESOBJECTSTRING.
** Address of variable length string
20 MQOD-RESOBJECTSTRING-VSPTR POINTER.
** Offset of variable length string
20 MQOD-RESOBJECTSTRING-VSOFFSET PIC S9(9) BINARY.
** size of buffer
20 MQOD-RESOBJECTSTRING-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQOD-RESOBJECTSTRING-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQOD-RESOBJECTSTRING-VSCCSID PIC S9(9) BINARY.
** Alias queue resolved object type
15 MQOD-RESOLVEDTYPE PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQOD based,
3 StructId          char(4),          /* Structure identifier */
3 Version           fixed bin(31), /* Structure version number */
3 ObjectType        fixed bin(31), /* Object type */
3 ObjectName        char(48),         /* Object name */
3 ObjectQMgrName    char(48),         /* Object queue manager name */
3 DynamicQName      char(48),         /* Dynamic queue name */
3 AlternateUserId   char(12),         /* Alternate user identifier */
3 RecsPresent       fixed bin(31), /* Number of object records
                                     present */
3 KnownDestCount    fixed bin(31), /* Number of local queues opened
                                     successfully */
3 UnknownDestCount  fixed bin(31), /* Number of remote queues opened
                                     successfully */
3 InvalidDestCount  fixed bin(31), /* Number of queues that failed to
                                     open */
3 ObjectRecOffset   fixed bin(31), /* Offset of first object record
                                     from start of MQOD */
3 ResponseRecOffset fixed bin(31), /* Offset of first response record
                                     from start of MQOD */

```

```

3 ObjectRecPtr      pointer,      /* Address of first object record */
3 ResponseRecPtr    pointer,      /* Address of first response
                                   record */
3 AlternateSecurityId char(40),    /* Alternate security identifier */
3 ResolvedQName     char(48),     /* Resolved queue name */
3 ResolvedQMgrName  char(48),     /* Resolved queue manager name */
3 ObjectString,     /* Object Long name */
5 VSPtr            pointer,      /* Address of variable length string */
5 VSOFFSET         fixed bin(31), /* Offset of variable length string */
5 VSBufSize        fixed bin(31), /* size of buffer */
5 VSLength         fixed bin(31), /* Length of variable length string */
5 VSCCSID          fixed bin(31), /* CCSID of variable length string */
3 SelectionString, /* Message Selection */
5 VSPtr            pointer,      /* Address of variable length string */
5 VSOFFSET         fixed bin(31), /* Offset of variable length string */
5 VSBufSize        fixed bin(31), /* size of buffer */
5 VSLength         fixed bin(31), /* Length of variable length string */
5 VSCCSID          fixed bin(31), /* CCSID of variable length string */
3 ResObjectString, /* Resolved Long object name */
5 VSPtr            pointer,      /* Address of variable length string */
5 VSOFFSET         fixed bin(31), /* Offset of variable length string */
5 VSBufSize        fixed bin(31), /* size of buffer */
5 VSLength         fixed bin(31), /* Length of variable length string */
5 VSCCSID          fixed bin(31), /* CCSID of variable length string */
3 ResolvedType     fixed bin(31); /* Alias queue resolved object type */

```

High Level Assembler declaration

```

MQOD                      DSECT
MQOD_STRUCID              DS    CL4   Structure identifier
MQOD_VERSION              DS    F     Structure version number
MQOD_OBJECTTYPE           DS    F     Object type
MQOD_OBJECTNAME           DS    CL48  Object name
MQOD_OBJECTQMGRNAME       DS    CL48  Object queue manager name
MQOD_DYNAMICQNAME         DS    CL48  Dynamic queue name
MQOD_ALTERNATEUSERID      DS    CL12  Alternate user identifier
MQOD_RECSPRESENT          DS    F     Number of object records present
MQOD_KNOWNDESTCOUNT      DS    F     Number of local queues opened
*                          successfully
MQOD_UNKNOWNDSTCOUNT     DS    F     Number of remote queues opened
*                          successfully
MQOD_INVALIDDESTCOUNT   DS    F     Number of queues that failed to
*                          open
MQOD_OBJECTRECOFFSET      DS    F     Offset of first object record from
*                          start of MQOD
MQOD_RESPONSERECOFFSET    DS    F     Offset of first response record
*                          from start of MQOD
MQOD_OBJECTRECPtr        DS    F     Address of first object record
MQOD_RESPONSERECPtr      DS    F     Address of first response record
MQOD_ALTERNATESECURITYID  DS    XL40  Alternate security identifier
MQOD_RESOLVEDQNAME        DS    CL48  Resolved queue name
MQOD_RESOLVEDQMGRNAME     DS    CL48  Resolved queue manager name
MQOD_OBJECTSTRING         DS    F     Object Long name
MQOD_OBJECTSTRING_VSPTR   DS    F     Address of variable length string
MQOD_OBJECTSTRING_VSOFFSET DS    F     Offset of variable length string
MQOD_OBJECTSTRING_VSBUSIZE DS    F     size of buffer
MQOD_OBJECTSTRING_VSLNGTH DS    F     Length of variable length string
MQOD_OBJECTSTRING_VSCSID  DS    F     CCSID of variable length string
MQOD_OBJECTSTRING_LENGTH EQU    *- MQOD_OBJECTSTRING
                                ORG    MQOD_OBJECTSTRING
MQOD_OBJECTSTRING_AREA   DS    CL(MQOD_OBJECTSTRING_LENGTH)
*
MQOD_SELECTIONSTRING     DS    F     Message Selector
MQOD_SELECTIONSTRING_VSPTR DS    F     Address of variable length string
MQOD_SELECTIONSTRING_VSOFFSET DS    F     Offset of variable length string
MQOD_SELECTIONSTRING_VSBUSIZE DS    F     size of buffer
MQOD_SELECTIONSTRING_VSLNGTH DS    F     Length of variable length string
MQOD_SELECTIONSTRING_VSCSID DS    F     CCSID of variable length string
MQOD_SELECTIONSTRING_LENGTH EQU    *- MQOD_SELECTIONSTRING
                                ORG    MQOD_SELECTIONSTRING
MQOD_SELECTIONSTRING_AREA DS    CL(MQOD_SELECTIONSTRING_LENGTH)
*
MQOD_RESOBJECTSTRING     DS    F     Resolved Long object name
MQOD_RESOBJECTSTRING_VSPTR DS    F     Address of variable length string
MQOD_RESOBJECTSTRING_VSOFFSET DS    F     Offset of variable length string
MQOD_RESOBJECTSTRING_VSBUSIZE DS    F     size of buffer
MQOD_RESOBJECTSTRING_VSLNGTH DS    F     Length of variable length string
MQOD_RESOBJECTSTRING_VSCSID DS    F     CCSID of variable length string
MQOD_RESOBJECTSTRING_LENGTH EQU    *- MQOD_RESOBJECTSTRING
                                ORG    MQOD_RESOBJECTSTRING

```

MQOD_RESOBJECTSTRING_AREA	DS	CL(MQOD_RESOBJECTSTRING_LENGTH)
MQOD_RESOLVEDTYPE	DS	F Alias queue object resolved type
*MQOD_LENGTH	EQU	*-MQOD
	ORG	MQOD
MQOD_AREA	DS	CL(MQOD_LENGTH)

Visual Basic declaration

```

Type MQOD
    StrucId          As String*4 'Structure identifier'
    Version          As Long      'Structure version number'
    ObjectType       As Long      'Object type'
    ObjectName       As String*48 'Object name'
    ObjectQMgrName   As String*48 'Object queue manager name'
    DynamicQName     As String*48 'Dynamic queue name'
    AlternateUserId  As String*12 'Alternate user identifier'
    RecsPresent      As Long      'Number of object records present'
    KnownDestCount   As Long      'Number of local queues opened'
                                'successfully'
    UnknownDestCount As Long      'Number of remote queues opened'
                                'successfully'
    InvalidDestCount As Long      'Number of queues that failed to'
                                'open'
    ObjectRecOffset  As Long      'Offset of first object record from'
                                'start of MQOD'
    ResponseRecOffset As Long      'Offset of first response record'
                                'from start of MQOD'
    ObjectRecPtr     As MQPTR     'Address of first object record'
    ResponseRecPtr   As MQPTR     'Address of first response record'
    AlternateSecurityId As MQBYTE40 'Alternate security identifier'
    ResolvedQName    As String*48 'Resolved queue name'
    ResolvedQMgrName As String*48 'Resolved queue manager name'
End Type

```

MQOR - Object record

The following table summarizes the fields in the structure.

Table 510. Fields in MQOR		
Field	Description	Topic
<i>ObjectName</i>	Object name	ObjectName
<i>ObjectQMgrName</i>	Object queue manager name	ObjectQMgrName

Overview for MQOR

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

Purpose: Use the MQOR structure to specify the queue name and queue-manager name of a single destination queue. MQOR is an input structure for the MQOPEN and MQPUT1 calls.

Character set and encoding: Data in MQOR must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQOPEN call, you can open a list of queues; this list is called a *distribution list*. Each message put using the queue handle returned by that MQOPEN call is placed on each of the queues in the list, provided that the queue was opened successfully.

Fields for MQOR

The MQOR structure contains the following fields; the fields are described in **alphabetical order**:

ObjectName (MQCHAR48)

This is the same as the *ObjectName* field in the MQOD structure (see MQOD for details), except that:

- It must be the name of a queue.
- It must not be the name of a model queue.

This is always an input field. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ObjectQMgrName (MQCHAR48)

This is the same as the *ObjectQMgrName* field in the MQOD structure (see MQOD for details).

This is always an input field. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

Initial values and language declarations for MQOR

Table 511. Initial values of fields in MQOR for MQOR		
Field name	Name of constant	Value of constant
<i>ObjectName</i>	None	Null string or blanks
<i>ObjectQMgrName</i>	None	Null string or blanks
Notes: <ol style="list-style-type: none">1. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages.2. In the C programming language, the macro variable MQOR_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure:<div><pre>MQOR MyOR = {MQOR_DEFAULT};</pre></div>		

C declaration

```
typedef struct tagMQOR MQOR;  
struct tagMQOR {  
    MQCHAR48  ObjectName;      /* Object name */  
    MQCHAR48  ObjectQMgrName; /* Object queue manager name */  
};
```

COBOL declaration

```
**  MQOR structure  
10 MQOR.  
**    Object name  
15 MQOR-OBJECTNAME    PIC X(48).  
**    Object queue manager name  
15 MQOR-OBJECTQMGRNAME PIC X(48).
```

PL/I declaration

```
dcl  
  1 MQOR based,  
    3 ObjectName      char(48), /* Object name */  
    3 ObjectQMgrName char(48); /* Object queue manager name */
```

Visual Basic declaration

```
Type MQOR  
    ObjectName As String*48 'Object name'
```

```
ObjectQMgrName As String*48 'Object queue manager name'  
End Type
```

MQPD - Property descriptor

The following table summarizes the fields in the structure.

Table 512. Fields in MQPD		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options
<i>Support</i>	Required support for message property	Support
<i>Context</i>	Message context to which property belongs	Context
<i>CopyOptions</i>	Copy options to which property belongs	CopyOptions

Overview for MQPD

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS and IBM MQ MQI clients.

Purpose: The **MQPD** is used to define the attributes of a property. The structure is an input/output parameter on the MQSETMP call and an output parameter on the MQINQMP call.

Character set and encoding: Data in **MQPD** must be in the character set of the application and encoding of the application (**MQENC_NATIVE**).

Fields for MQPD

The MQPD structure contains the following fields; the fields are described in **alphabetical order**:

Context (MQLONG)

This describes what message context the property belongs to.

When a queue manager receives a message containing an IBM MQ-defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *Context* field.

The following option can be specified:

MQPD_USER_CONTEXT

The property is associated with the user context.

No special authorization is required to be able to set a property associated with the user context using the MQSETMP call.

On a IBM WebSphere® MQ 7.0 queue manager, a property associated with the user context is saved as described for MQOO_SAVE_ALL_CONTEXT. An MQPUT call with MQPMO_PASS_ALL_CONTEXT specified, causes the property to be copied from the saved context into the new message.

If the option previously described is not required, the following option can be used:

MQPD_NO_CONTEXT

The property is not associated with a message context.

An unrecognized value is rejected with a *Reason* code of MQRC_PD_ERROR

This is an input/output field to the MQSETMP call and an output field from the MQINQMP call. The initial value of this field is MQPD_NO_CONTEXT.

CopyOptions (MQLONG)

This describes which type of messages the property should be copied into. This is an output only field for recognized IBM MQ defined properties; IBM MQ sets the appropriate value.

When a queue manager receives a message containing an IBM MQ defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *CopyOptions* field.

You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

MQCOPY_FORWARD

This property is copied into a message being forwarded.

MQCOPY_PUBLISH

This property is copied into the message received by a subscriber when a message is being published.

MQCOPY_REPLY

This property is copied into a reply message.

MQCOPY_REPORT

This property is copied into a report message.

MQCOPY_ALL

This property is copied into all types of subsequent messages.

Default option: The following option can be specified to supply the default set of copy options:

MQCOPY_DEFAULT

This property is copied into a message being forwarded, into a report message, or into a message received by a subscriber when a message is being published.

This is equivalent to specifying the combination of options MQCOPY_FORWARD, plus MQCOPY_REPORT, plus MQCOPY_PUBLISH.

If none of the options that are described previously is required, use the following option:

MQCOPY_NONE

Use this value to indicate that no other copy options are specified; programmatically no relationship exists between this property and subsequent messages. This is always returned for message descriptor properties.

This is an input/output field to the MQSETMP call and an output field from the MQINQMP call. The initial value of this field is MQCOPY_DEFAULT.

Options (MQLONG)

The value must be:

MQPD_NONE

No options specified

This is always an input field. The initial value of this field is MQPD_NONE.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQPD_STRUC_ID

Identifier for property descriptor structure.

For the C programming language, the constant **MQPD_STRUC_ID_ARRAY** is also defined; this has the same value as **MQPD_STRUC_ID**, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is **MQPD_STRUC_ID**.

Support (MQLONG)

This field describes what level of support for the message property is required of the queue manager, in order for the message containing this property to be put to a queue. This applies only to IBM MQ-defined properties; support for all other properties is optional.

The field is automatically set to the correct value when the IBM MQ-defined property is known by the queue manager. If the property is not recognized, MQPD_SUPPORT_OPTIONAL is assigned. When a queue manager receives a message containing an IBM MQ-defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *Support* field.

When setting an IBM MQ-defined property using the MQSETMP call on a message handle where the MQCMHO_NO_VALIDATION option was set, *Support* becomes an input field. This allows an application to put an IBM MQ-defined property, with the correct value, where the property is unsupported by the connected queue manager, but where the message is intended to be processed on another queue manager.

The value MQPD_SUPPORT_OPTIONAL is always assigned to properties that are not IBM MQ-defined properties.

If a IBM WebSphere MQ 7.0 queue manager, that supports message properties, receives a property that contains an unrecognized *Support* value, the property is treated as if:

- MQPD_SUPPORT_REQUIRED was specified if any of the unrecognized values are contained in the MQPD_REJECT_UNSUP_MASK.
- MQPD_SUPPORT_REQUIRED_IF_LOCAL was specified if any of the unrecognized values are contained in the MQPD_ACCEPT_UNSUP_IF_XMIT_MASK
- MQPD_SUPPORT_OPTIONAL was specified otherwise.

One of the following values is returned by the MQINQMP call, or one of the values can be specified, when using the MQSETMP call on a message handle where the MQCMHO_NO_VALIDATION option is set:

MQPD_SUPPORT_OPTIONAL

The property is accepted by a queue manager even if it is not supported. The property can be discarded in order for the message to flow to a queue manager that does not support message properties. This value is also assigned to properties that are not IBM MQ-defined.

MQPD_SUPPORT_REQUIRED

Support for the property is required. The message is rejected by a queue manager that does not support the IBM MQ-defined property. The MQPUT or MQPUT1 call fails with completion code MQCC_FAILED and reason code MQRC_UNSUPPORTED_PROPERTY.

MQPD_SUPPORT_REQUIRED_IF_LOCAL

The message is rejected by a queue manager that does not support the IBM MQ-defined property if the message is destined for a local queue. The MQPUT or MQPUT1 call fails with completion code MQCC_FAILED and reason code MQRC_UNSUPPORTED_PROPERTY.

The MQPUT or MQPUT1 call succeeds if the message is destined for a remote queue manager.

This is an output field on the MQINQMP call and an input field on the MQSETMP call if the message handle was created with the MQCMHO_NO_VALIDATION option set. The initial value of this field is MQPD_SUPPORT_OPTIONAL.

Version (MQLONG)

This is the structure version number; the value must be:

MQPD_VERSION_1

Version-1 property descriptor structure.

The following constant specifies the version number of the current version:

MQPD_CURRENT_VERSION

Current version of property descriptor structure.

This is always an input field. The initial value of this field is **MQPD_VERSION_1**.

Initial values and language declarations for MQPD

Table 513. Initial values of fields in MQPD		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQPD_STRUC_ID	'PD'
<i>Version</i>	MQPD_VERSION_1	1
<i>Options</i>	MQPD_NONE	0
<i>Support</i>	MQPD_SUPPORT_OPTIONAL	0
<i>Context</i>	MQPD_NO_CONTEXT	0
<i>CopyOptions</i>	MQCOPY_DEFAULT	0
Notes: 1. In the C programming language, the macro variable MQPD_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQPD MyPD = {MQPD_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQPD MQPD;
struct tagMQPD {
    MQCHAR4  StrucId;      /* Structure identifier */
    MQLONG   Version;      /* Structure version number */
    MQLONG   Options;      /* Options that control the action of
                           MQSETMP and MQINQMP */
    MQLONG   Support;      /* Property support option */
    MQLONG   Context;      /* Property context */
    MQLONG   CopyOptions; /* Property copy options */
};
```

COBOL declaration

```
**      MQPD structure
10 MQPD.
**      Structure identifier
15 MQPD-STRUCID PIC X(4).
**      Structure version number
15 MQPD-VERSION PIC S9(9) BINARY.
**      Options that control the action of MQSETMP and
**      MQINQMP
15 MQPD-OPTIONS PIC S9(9) BINARY.
**      Property support option
15 MQPD-SUPPORT PIC S9(9) BINARY.
**      Property context
15 MQPD-CONTEXT PIC S9(9) BINARY.
**      Property copy options
15 MQPD-COPYOPTIONS PIC S9(9) BINARY.
```

PL/I declaration

```
dcl
1 MQPD based,
3 StrucId      char(4),      /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 Options      fixed bin(31), /* Options that control the action
                             of MQSETMP and MQINQMP */
3 Support      fixed bin(31), /* Property support option */
3 Context      fixed bin(31), /* Property context */
3 CopyOptions  fixed bin(31); /* Property copy options */
```

High Level Assembler declaration

```

MQPD          DSECT
MQPD_STRUCID  DS   CL4    Structure identifier
MQPD_VERSION  DS   F      Structure version number
MQPD_OPTIONS  DS   F      Options that control the
*              action of MQSETMP and MQINQMP
MQPD_SUPPORT  DS   F      Property support option
MQPD_CONTEXT  DS   F      Property context
MQPD_COPYOPTIONS DS   F    Property copy options
MQPD_LENGTH   EQU   *-MQPD
MQPD_AREA     DS      CL(MQPD_LENGTH)

```

MQPMO - Put-message options

The following table summarizes the fields in the structure.

Table 514. MQPMO structure		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options that control the action of MQPUT and MQPUT1	Options
<i>Timeout</i>	Reserved	Timeout
<i>Context</i>	Object handle of input queue	Context
<i>KnownDestCount</i>	Number of messages sent successfully to local queues	KnownDestCount
<i>UnknownDestCount</i>	Number of messages sent successfully to remote queues	UnknownDestCount
<i>InvalidDestCount</i>	Number of messages that could not be sent	InvalidDestCount
<i>ResolvedQName</i>	Resolved name of destination queue	ResolvedQName
<i>ResolvedQMgrName</i>	Resolved name of destination queue manager	ResolvedQMgrName
Note: The remaining fields are ignored if <i>Version</i> is less than MQPMO_VERSION_2.		
<i>RecsPresent</i>	Number of put message records or response records present	RecsPresent
<i>PutMsgRecFields</i>	Flags indicating which MQPMR fields are present	PutMsgRecFields
<i>PutMsgRecOffset</i>	Offset of first put-message record from start of MQPMO	PutMsgRecOffset
<i>ResponseRecOffset</i>	Offset of first response record from start of MQPMO	ResponseRecOffset
<i>PutMsgRecPtr</i>	Address of first put message record	PutMsgRecPtr
<i>ResponseRecPtr</i>	Address of first response record	ResponseRecPtr
Note: The remaining fields are ignored if <i>Version</i> is less than MQPMO_VERSION_3.		
<i>OriginalMsgHandle</i>	Original message handle	OriginalMsgHandle
<i>NewMsgHandle</i>	New message handle	NewMsgHandle

Table 514. MQPMO structure (continued)		
Field	Description	Topic
<i>Action</i>	Type of put being performed and the relationship between the original message specified by the <i>OriginalMsgHandle</i> field and the new message specified by the <i>NewMsgHandle</i> field	Action
<i>PubLevel</i>	Level of subscription targeted by the publication	PubLevel

Overview for MQPMO

Availability: All IBM MQ systems, plus IBM MQ clients connected to these systems.

Purpose: The MQPMO structure allows the application to specify options that control how messages are placed on queues, or published to topics. The structure is an input/output parameter on the MQPUT and MQPUT1 calls.

Version: The current version of MQPMO is MQPMO_VERSION_3. Certain fields are available only in certain versions of MQPMO. If you need to port applications between several environments, you must ensure that the version of MQPMO is consistent across all environments. Fields that exist only in particular versions of the structure are identified as such in [“MQPMO - Put-message options” on page 475](#) and in the field descriptions.

The header, COPY, and INCLUDE files provided for the supported programming languages contain the most-recent version of MQPMO that is supported by the environment, but with the initial value of the *Version* field set to MQPMO_VERSION_1. To use fields that are not present in the version-1 structure, the application must set the *Version* field to the version number of the version required.

Character set and encoding: Data in MQPMO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQPMO

The MQPMO structure contains the following fields; the fields are described in **alphabetical order**:

Action (MQLONG)

This specifies the type of put being performed and the relationship between the original message specified by the *OriginalMsgHandle* field and the new message specified by the *NewMsgHandle* field. The properties of the message are chosen by the queue manager according to the value of the *Action* specified.

You can choose to supply the contents of the message descriptor using the *MsgDesc* parameter on the MQPUT or MQPUT1 calls. Alternatively it is possible not to supply the *MsgDesc* parameter, or to specify that it is output-only by including MQPMO_MD_FOR_OUTPUT_ONLY in the *Options* field of the MQPMO structure.

If the *MsgDesc* parameter is not supplied, or if it is specified to be output-only, then the message descriptor for the new message is populated from the message handle fields of the MQPMO, according to the rules described in this topic.

The context setting and passing activities described in [Controlling context information](#) take effect after the message descriptor has been composed.

If an incorrect action value is specified, the call fails with the reason code MQRC_ACTION_ERROR.

Any one of the following actions can be specified:

MQACTP_NEW

A new message is being put, and no relationship to a previous message is being specified by the program. The message descriptor is composed as follows:

- If a MsgDesc is supplied on the MQPUT or MQPUT1 call, and MQPMO_MD_FOR_OUTPUT_ONLY is not in the MQPMO.Options, this is used as the message descriptor unmodified.
- If a MsgDesc is not supplied, or MQPMO_MD_FOR_OUTPUT_ONLY is in the MQPMO.Options then the queue manager generates the message descriptor using a combination of properties from OriginalMsgHandle and NewMsgHandle. Any message descriptor fields explicitly set on the new message handle take precedence over those in the original message handle.

Message data is taken from the MQPUT or MQPUT1 Buffer parameter.

MQACTP_FORWARD

A previously retrieved message is being forwarded. The original message handle specifies the message that was previously retrieved.

The new message handle specifies any modifications to the properties (including any in the message descriptor) in the original message handle.

The message descriptor is composed as follows:

- If a MsgDesc is supplied on the MQPUT or MQPUT1 call, and MQPMO_MD_FOR_OUTPUT_ONLY is not in the MQPMO.Options, this is used as the message descriptor unmodified.
- If a MsgDesc is not supplied, or MQPMO_MD_FOR_OUTPUT_ONLY is in the MQPMO.Options then the queue manager generates the message descriptor using a combination of properties from OriginalMsgHandle and NewMsgHandle. Any message descriptor fields explicitly set on the new message handle take precedence over those in the original message handle.
- If MQPMO_NEW_MSG_ID or MQPMO_NEW_CORREL_ID are specified in the MQPMO.Options, then these are honoured.

The message properties are composed as follows:

- All properties from the original message handle which have MQCOPY_FORWARD in the MQPD.CopyOptions
- All properties from the new message handle. For each property in the new message handle that has the same name as a property in the original message handle, the value is taken from the new message handle. The only exception to this rule is the special case when the property in the new message handle has the same name as a property in the original message handle, but the value of the property is null. In this case the property is removed from the message.

The message data to be forwarded is taken from the MQPUT or MQPUT1 Buffer parameter.

MQACTP_REPLY

A reply is being made to a previously retrieved message. The original message handle specifies the message that was previously retrieved.

The new message handle specifies any modifications to the properties (including any in the message descriptor) in the original message handle.

The message descriptor is composed as follows:

- If a MsgDesc is supplied on the MQPUT or MQPUT1 call, and MQPMO_MD_FOR_OUTPUT_ONLY is not in the MQPMO.Options, this is used as the message descriptor unmodified.
- If a MsgDesc is not supplied, or MQPMO_MD_FOR_OUTPUT_ONLY is in the MQPMO.Options then initial message descriptor fields are chosen as follows:

Table 515. Reply message handle transformation	
Field in MQMD	Value used
Report	If MQRO_PASS_DISCARD_AND_EXPIRY and MQRO_DISCARD_MSG are set: MQRO_DISCARD_MSG otherwise MQRO_NONE
MsgType	MQMT_REPLY
Expiry	If MQRO_PASS_DISCARD_AND_EXPIRY is set: Copied from the input message otherwise MQEI_UNLIMITED
Feedback	MQFB_NONE
MsgId	If MQPMO_NEW_MSG_ID is set: A new message identifier is generated else if MQRO_PASS_MSG_ID is set: Copied from the input message otherwise MQMI_NONE
CorrelId	If MQPMO_NEW_CORREL_ID is set: A new correlation identifier is generated else if MQRO_COPY_MSG_ID_TO_CORREL_ID is set: Copied from the MsgId field of the input message else if MQRO_PASS_CORREL_ID is set: Copied from the CorrelId field of the input message otherwise MQCI_NONE
BackoutCount	0
ReplyToQ	Blanks
ReplyToQMgr	Blanks
GroupId	MQGI_NONE
MsgSeqNumber	1
Offset	0
MsgFlags	MQMF_NONE
OriginalLength	MQOL_UNDEFINED

- The message descriptor is then modified by the new message handle - any message descriptor fields explicitly set as properties in the new message handle take precedence over the message descriptor fields as described previously.

The message properties are composed as follows:

- All properties from the original message handle which have MQCOPY_REPLY in the MQPD.CopyOptions

- All properties from the new message handle. For each property in the new message handle that has the same name as a property in the original message handle, the value is taken from the new message handle. The only exception to this rule is the special case when the property in the new message handle has the same name as a property in the original message handle, but the value of the property is null. In this case the property is removed from the message.

The message data to be forwarded is taken from the MQPUT/MQPUT1 Buffer parameter.

MQACTP_REPORT

A report is being generated as a result of a previously retrieved message. The original message handle specifies the message causing the report to be generated.

The new message handle specifies any modifications to the properties (including any in the message descriptor) in the original message handle.

The message descriptor is composed as follows:

- If a MsgDesc is supplied on the MQPUT or MQPUT1 call, and MQPMO_MD_FOR_OUTPUT_ONLY is not in the MQPMO.Options, this is used as the message descriptor unmodified.
- If a MsgDesc is not supplied, or MQPMO_MD_FOR_OUTPUT_ONLY is in the MQPMO.Options then initial message descriptor fields are chosen as follows:

<i>Table 516. Report message handle transformation</i>	
Field in MQMD	Value used
Report	If MQRO_PASS_DISCARD_AND_EXPIRY and MQRO_DISCARD_MSG are set: MQRO_DISCARD_MSG otherwise MQRO_NONE
MsgType	MQMT_REPORT
Expiry	If MQRO_PASS_DISCARD_AND_EXPIRY is set: Copied from the input message otherwise MQEI_UNLIMITED
MsgId	If MQPMO_NEW_MSG_ID is set: A new message identifier is generated else if MQRO_PASS_MSG_ID is set: Copied from the input message otherwise MQMI_NONE
CorrelId	If MQPMO_NEW_CORREL_ID is set: A new correlation identifier is generated else if MQRO_COPY_MSG_ID_TO_CORREL_ID is set: Copied from the MsgId field of the input message else if MQRO_PASS_CORREL_ID is set: Copied from the CorrelId field of the input message otherwise MQCI_NONE
BackoutCount	0

Table 516. Report message handle transformation (continued)	
Field in MQMD	Value used
ReplyToQ	Blanks
ReplyToQMGr	Blanks
OriginalLength	Set to the <i>BufferLength</i>

- The message descriptor is then modified by the new message handle - any message descriptor fields explicitly set as properties in the new message handle take precedence over the message descriptor fields as described previously.

The message properties are composed as follows:

- All properties from the original message handle which have MQCOPY_REPORT in the MQPD.CopyOptions
- All properties from the new message handle. For each property in the new message handle that has the same name as a property in the original message handle, the value is taken from the new message handle. The only exception to this rule is the special case when the property in the new message handle has the same name as a property in the original message handle, but the value of the property is null. In this case the property is removed from the message.

The Feedback field in the resultant MQMD represents the report that is to be generated. A Feedback value of MQFB_NONE causes the MQPUT or MQPUT1 call to fail with reason code MQRC_FEEDBACK_ERROR.

To choose the user data of the report message, IBM MQ consults the Report and Feedback fields in the resultant MQMD, and the Buffer and BufferLength parameters of the MQPUT or MQPUT1 call.

- If Feedback is MQFB_COA, MQFB_COD or MQFB_EXPIRATION then the value of Report is inspected.
- If any of the following cases is true, the full message data from Buffer for a length of BufferLength is used.
 - Feedback is MQFB_EXPIRATION and Report contains MQRO_EXPIRATION_WITH_FULL_DATA
 - Feedback is MQFB_COD and Report contains MQRO_COD_WITH_FULL_DATA
 - Feedback is MQFB_COA and Report contains MQRO_COA_WITH_FULL_DATA
- If any of the following cases is true, the first 100 bytes of the message (or BufferLength if this is less than 100) from Buffer are used
 - Feedback is MQFB_EXPIRATION and Report contains MQRO_EXPIRATION_WITH_DATA
 - Feedback is MQFB_COD and Report contains MQRO_COD_WITH_DATA
 - Feedback is MQFB_COA and Report contains MQRO_COA_WITH_DATA
- If Feedback is MQFB_EXPIRATION, MQFB_COD or MQFB_COA, and Report does not contain the *_WITH_FULL_DATA or *_WITH_DATA options relevant to that Feedback value, then no user data is included with the message.
- If Feedback takes a different value from those listed above, then Buffer and BufferLength are used as normal.

The derivation of the user data is shown in the following table:

Context (MQHOBJ)

If MQPMO_PASS_IDENTITY_CONTEXT or MQPMO_PASS_ALL_CONTEXT is specified, this field must contain the input queue handle from which context information to be associated with the message being put is taken.

If neither MQPMO_PASS_IDENTITY_CONTEXT nor MQPMO_PASS_ALL_CONTEXT is specified, this field is ignored.

This is an input field. The initial value of this field is 0.

InvalidDestCount (MQLONG)

This is the number of messages that could not be sent to queues in the distribution list. The count includes queues that failed to open, as well as queues that were opened successfully but for which the put operation failed. This field is also set when putting a message to a single queue that is not in a distribution list.

Note: This field is set if the *CompCode* parameter on the MQPUT or MQPUT1 call is MQCC_OK or MQCC_WARNING; it might be set if the *CompCode* parameter is MQCC_FAILED, but do not rely on this in application code.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_1.

This field is undefined on z/OS because distribution lists are not supported.

KnownDestCount (MQLONG)

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that are local queues. The count does not include messages sent to queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). This field is also set when putting a message to a single queue that is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_1.

This field is undefined on z/OS because distribution lists are not supported.

NewMsgHandle (MQHMSG)

This is an optional handle to the message being put subject to the value of the Action field. It defines the properties of the message and overrides the values of the *OriginalMsgHandle*, if specified.

On return from the **MQPUT** or **MQPUT1** call, the contents of the handle reflect the message that was actually put.

This is an input field. The initial value of this field is **MQHM_NONE**. This field is ignored if *Version* is less than **MQPMO_VERSION_3**.

MQPMO options (MQLONG)

The Options field controls the operation of **MQPUT** and **MQPUT1** calls.

Scope option. You can specify any or none of the MQPMO options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations). Combinations that are not valid are noted; any other combinations are valid.

The following option controls the scope of the publications sent:

MQPMO_SCOPE_QMGR

The publication is sent only to subscribers that have subscribed on this queue manager. The publication is not forwarded to any remote publish/subscribe queue managers that have made a subscription to this queue manager, which overrides any behavior that has been set using the PUBSCOPE topic attribute.

Note: If not set, the publication scope is determined by the PUBSCOPE topic attribute.

Publishing options. The following options control the way messages are published to a topic:

MQPMO_SUPPRESS_REPLYTO

Any information specified in the *ReplyToQ* and *ReplyToQMGr* fields of the MQMD of this publication is not passed on to subscribers. If this option is used with a report option that requires a *ReplyToQ*, the call fails with MQRC_MISSING_REPLY_TO_Q.

MQPMO_RETAIN

The publication being sent is to be retained by the queue manager. This retention allows a subscriber to request a copy of this publication after the time it was published, by using the MQSUBRQ call. It also allows a publication to be sent to applications which make their subscription after the time this publication was made (unless they choose not to be sent it by using the option MQSO_NEW_PUBLICATIONS_ONLY). If an application is sent a publication which was retained, it is indicated by the MQIsRetained message property of that publication.

Only one publication can be retained at each node of the topic tree. Therefore, if there already is a retained publication for this topic, published by any other application, it is replaced with this publication. It is therefore better to avoid having more than one publisher retaining messages on the same topic.

When retained publications are requested by a subscriber, the subscription used might contain a wildcard in the topic, in which case a number of retained publications might match (at various nodes in the topic tree) and several publications might be sent to the requesting application. See the description of the [“MQSUBRQ - Subscription request”](#) on page 765 call for more details.

For information about how retained publications interact with subscription levels, see [Intercepting publications](#).

If this option is used and the publication cannot be retained, the message is not published and the call fails with MQRC_PUT_NOT_RETAINED.

MQPMO_NOT_OWN_SUBS

Tells the queue manager that the application does not want to send any of its publications to subscriptions it owns. Subscriptions are considered to be owned by the same application if the connection handles are the same.

MQPMO_WARN_IF_NO_SUBS_MATCHED

If no subscription matches the publication, return a completion code (*CompCode*) of MQCC_WARNING and reason code MQRC_NO_SUBS_MATCHED.

If MQRC_NO_SUBS_MATCHED is returned by the put operation, the publication was not delivered to any subscriptions. However, if the MQPMO_RETAIN option is specified on the put operation, the message is retained and delivered to any subsequently defined matching subscription.

A subscription on the topic matches the publication if any of the following conditions are met:

- The message is delivered to the subscription queue
- The message would have been delivered to the subscription queue but a problem with the queue means that the message cannot be put to the queue, and it was consequently placed on the dead letter queue or discarded.
- A routing exit is defined that suppresses delivery of the message to the subscription

A subscription on the topic does not match the publication if any of the following conditions are met:

- The subscription has a selection string that does not match the publication
- The subscription specified the MQSO_PUBLICATION_ON_REQUEST option
- The publication is not delivered because the MQPMO_NOT_OWN_SUBS option was specified on the put operation and the subscription matches the identity of the publisher

Syncpoint options. The following options relate to the participation of the MQPUT or MQPUT1 call within a unit of work:

MQPMO_SYNCPOINT

The request is to operate within the normal unit-of-work protocols. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted.

If MQPMO_SYNCPOINT and MQPMO_NO_SYNCPOINT are not specified, the inclusion of the put request in unit-of-work protocols is determined by the environment running the queue manager and

not the environment running the application. On z/OS, the put request is within a unit of work. In all other environments, the put request is not within a unit of work.

Because of these differences, an application that you want to port must not allow this option to default; specify either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT explicitly.

Do not specify MQPMO_SYNCPOINT with MQPMO_NO_SYNCPOINT.

MQPMO_NO_SYNCPOINT

The request is to operate outside the normal unit-of-work protocols. The message is available immediately, and it cannot be deleted by backing out a unit of work.

If MQPMO_NO_SYNCPOINT and MQPMO_SYNCPOINT are not specified, the inclusion of the put request in unit-of-work protocols is determined by the environment running the queue manager and not the environment running the application. On z/OS, the put request is within a unit of work. In all other environments, the put request is not within a unit of work.

Because of these differences, an application that you want to port must not allow this option to default; specify either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT explicitly.

Do not specify MQPMO_NO_SYNCPOINT with MQPMO_SYNCPOINT.

Message-identifier and correlation-identifier options. The following options request the queue manager to generate a new message identifier or correlation identifier:

MQPMO_NEW_MSG_ID

The queue manager replaces the contents of the *MsgId* field in MQMD with a new message identifier. This message identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

The MQPMO_NEW_MSG_ID option can also be specified when the message is being put to a distribution list; see the description of the *MsgId* field in the MQPMR structure for details.

Using this option relieves the application of the need to reset the *MsgId* field to MQMI_NONE before each MQPUT or MQPUT1 call.

MQPMO_NEW_CORREL_ID

The queue manager replaces the contents of the *CorrelId* field in MQMD with a new correlation identifier. This correlation identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

The MQPMO_NEW_CORREL_ID option can also be specified when the message is being put to a distribution list; see the description of the *CorrelId* field in the MQPMR structure for details.

MQPMO_NEW_CORREL_ID is useful in situations where the application requires a unique correlation identifier.

Group and segment options. The following options relate to the processing of messages in groups and segments of logical messages. Read the definitions that follow to help you to understand the option.



Attention: You cannot use segmented or grouped messages with Publish/Subscribe.

Physical message

This is the smallest unit of information that can be placed on or removed from a queue; it often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor (MQMD). Generally, physical messages are distinguished by differing values for the message identifier (*MsgId* field in MQMD), although this is not enforced by the queue manager.

Logical message

A logical message is a single unit of application information for non- z/OS platforms only. In the absence of system constraints, a logical message is the same as a physical message. But where logical messages are extremely large, system constraints might make it advisable or necessary to split a logical message into two or more physical messages, called *segments*.

A logical message that has been segmented consists of two or more physical messages that have the same non-null group identifier (*GroupId* field in MQMD), and the same message sequence number (*MsgSeqNumber* field in MQMD). The segments are distinguished by differing values for the segment offset (*Offset* field in MQMD), which gives the offset of the data in the physical message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message usually have differing message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a non-null group identifier, although in this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages for which segmentation has been inhibited by the sending application have a null group identifier (MQGI_NONE), unless the logical message belongs to a message group.

Message group

A message group is a set of one or more logical messages that have the same non-null group identifier. The logical messages in the group are distinguished by differing values for the message sequence number, which is an integer in the range 1 through *n*, where *n* is the number of logical messages in the group. If one or more of the logical messages is segmented, there are more than *n* physical messages in the group.

MQPMO_LOGICAL_ORDER

This option tells the queue manager how the application puts messages in groups and segments of logical messages. It can be specified only on the MQPUT call; it is not valid on the MQPUT1 call.

If MQPMO_LOGICAL_ORDER is specified, it indicates that the application uses successive MQPUT calls to:

1. Put the segments in each logical message in the order of increasing segment offset, starting from 0, with no gaps.
2. Put all the segments in one logical message before putting the segments in the next logical message.
3. Put the logical messages in each message group in the order of increasing message sequence number, starting from 1, with no gaps. IBM MQ increments the message sequence number automatically.
4. Put all the logical messages in one message group before putting logical messages in the next message group.

For detailed information about MQPMO_LOGICAL_ORDER, see [Logical and physical ordering](#)

Context options. The following options control the processing of message context:

MQPMO_NO_CONTEXT

Both identity and origin context are set to indicate no context. This means that the context fields in MQMD are set to:

- Blanks for character fields
- Nulls for byte fields
- Zeros for numeric fields

MQPMO_DEFAULT_CONTEXT

The message is to have default context information associated with it, for both identity and origin. The queue manager sets the context fields in the message descriptor as follows:

Field in MQMD	Value used
<i>UserIdentifier</i>	Determined from the environment if possible; set to blanks otherwise.
<i>AccountingToken</i>	Determined from the environment if possible; set to MQACT_NONE otherwise.
<i>ApplIdentityData</i>	Set to blanks.

Field in MQMD	Value used
<i>PutApplType</i>	Determined from the environment.
<i>PutApplName</i>	Determined from the environment if possible; set to blanks otherwise.
<i>PutDate</i>	Set to the date when message is put.
<i>PutTime</i>	Set to the time when message is put.
<i>ApplOriginData</i>	Set to blanks.

For more information about message context, see [Message context](#).

These are the default values and actions if no context options are specified.

MQPMO_PASS_IDENTITY_CONTEXT

The message is to have context information associated with it. Identity context is taken from the queue handle specified in the *Context* field. Origin context information is generated by the queue manager in the same way that it is for MQPMO_DEFAULT_CONTEXT (see the preceding table for values). For more information about message context, see [Message context](#).

For the MQPUT call, the queue must have been opened with the MQOO_PASS_IDENTITY_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_PASS_IDENTITY_CONTEXT option.

MQPMO_PASS_ALL_CONTEXT

The message is to have context information associated with it. Context is taken from the queue handle specified in the *Context* field. For more information about message context, see [Controlling context information](#).

For the MQPUT call, the queue must have been opened with the MQOO_PASS_ALL_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_PASS_ALL_CONTEXT option.

MQPMO_SET_IDENTITY_CONTEXT

The message is to have context information associated with it. The application specifies the identity context in the MQMD structure. Origin context information is generated by the queue manager in the same way that it is for MQPMO_DEFAULT_CONTEXT (see the preceding table for values). For more information about message context, see [Message context](#).

For the MQPUT call, the queue must have been opened with the MQOO_SET_IDENTITY_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_SET_IDENTITY_CONTEXT option.

MQPMO_SET_ALL_CONTEXT

The message is to have context information associated with it. The application specifies the identity, origin, and user context in the MQMD structure. For more information about message context, see [Message context](#).

For the MQPUT call, the queue must have been opened with the MQOO_SET_ALL_CONTEXT option. For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_SET_ALL_CONTEXT option.

You can specify only one of the MQPMO_*_CONTEXT context options. If you specify none, MQPMO_DEFAULT_CONTEXT is assumed.

Property options. The following option relates to the properties of the message:

MQPMO_MD_FOR_OUTPUT_ONLY

The message descriptor parameter must only be used for output to return the message descriptor of the message that was put. The message descriptor fields associated with the *NewMsgHandle*, *OriginalMsgHandle*, or both fields, of the **MQPMO** structure must be used for input.

If a valid message handle is not provided then the call fails with reason code **MQRC_MD_ERROR**.

Put response options. The following options control the response returned to an MQPUT or MQPUT1 call. You can specify only one of these options. If MQPMO_ASYNC_RESPONSE and MQPMO_SYNC_RESPONSE are not specified, MQPMO_RESPONSE_AS_Q_DEF or MQPMO_RESPONSE_AS_TOPIC_DEF is assumed.

MQPMO_ASYNC_RESPONSE

The MQPMO_ASYNC_RESPONSE option requests that an MQPUT or MQPUT1 operation is completed without the application waiting for the queue manager to complete the call. Using this option can improve messaging performance, particularly for applications using client bindings. An application can periodically check, using the MQSTAT verb, whether an error has occurred during any previous asynchronous calls.

With this option, only the following fields are guaranteed to be completed in the MQMD;

- ApplIdentityData
- PutApplType
- PutApplName
- ApplOriginData

Additionally, if either or both of MQPMO_NEW_MSG_ID or MQPMO_NEW_CORREL_ID are specified as options, the MsgId and CorrelId returned are also completed. (MQPMO_NEW_MSG_ID can be implicitly specified by specifying a blank MsgId field).

Only the preceding specified fields are completed. Other information that would normally be returned in the MQMD or MQPMO structure is undefined.

When requesting asynchronous put response for MQPUT1, the ResolvedQName and ResolvedQMgrName returned in the MQOD structure are undefined.

When requesting asynchronous put response for MQPUT or MQPUT1, a CompCode and Reason of MQCC_OK and MQRC_NONE does not necessarily mean that the message was successfully put to a queue. When developing an MQI application that uses asynchronous put response and requires confirmation that messages have been put to a queue you must check both CompCode and Reason codes from the put operations and also use MQSTAT to query asynchronous error information.

Although the success or failure of each individual MQPUT or MQPUT1 call might not be returned immediately, the first error that occurred under an asynchronous call can be determined later through a call to MQSTAT.

If a persistent message under syncpoint fails to be delivered using asynchronous put response, and you attempt to commit the transaction, the commit fails and the transaction is backed out with a completion code of MQCC_FAILED and a reason of MQRC_BACKED_OUT. The application can make a call to MQSTAT to determine the cause of a previous MQPUT or MQPUT1 failure.

MQPMO_SYNC_RESPONSE

Specifying this put response type ensures that the MQPUT or MQPUT1 operation is always issued synchronously. If the put operation is successful, all fields in the MQMD and MQPMO are completed.

This option ensures a synchronous response irrespective of the default put response value defined on the queue or topic object.

MQPMO_RESPONSE_AS_Q_DEF

If this value is specified for an MQPUT call, the put response type used is taken from the DEFPRESP value specified on the queue when it was first opened by the application. If a client application is connected to a queue manager at a level earlier than Version 7.0, it behaves as if MQPMO_SYNC_RESPONSE was specified.

If this option is specified for an MQPUT1 call, the value of the DEFPRESP attribute is not known before the request is sent to the server. By default, if the MQPUT1 call is using MQPMO_SYNCPOINT it behaves as for MQPMO_ASYNC_RESPONSE, and if it is using MQPMO_NO_SYNCPOINT it behaves as for MQPMO_SYNC_RESPONSE. However, you can override this default behavior by setting the Put1DefaultAlwaysSync property in the client configuration file, see [CHANNELS stanza of the client configuration file](#).

MQPMO_RESPONSE_AS_TOPIC_DEF

MQPMO_RESPONSE_AS_TOPIC_DEF is a synonym for MQPMO_RESPONSE_AS_Q_DEF for use with topic objects.

Other options. The following options control authorization checking, what happens when the queue manager is quiescing, and resolving queue and queue manager names:

MQPMO_ALTERNATE_USER_AUTHORITY

MQPMO_ALTERNATE_USER_AUTHORITY indicates that the *AlternateUserId* field in the *ObjDesc* parameter of the MQPUT1 call contains a user identifier that is to be used to validate authority to put messages on the queue. The call can succeed only if *AlternateUserId* is authorized to open the queue with the specified options, regardless of whether the user identifier under which the application is running is authorized to do so. (This does not apply to the context options specified, however, which are always checked against the user identifier under which the application is running.)

This option is valid only with the MQPUT1 call.

MQPMO_FAIL_IF QUIESCING

This option forces the MQPUT or MQPUT1 call to fail if the queue manager is in the quiescing state.

On z/OS, this option also forces the MQPUT or MQPUT1 call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

The call returns completion code MQCC_FAILED with reason code MQRC_Q_MGR_QUIESCING or MQRC_CONNECTION_QUIESCING.

MQPMO_RESOLVE_LOCAL_Q

Use this option to fill *ResolvedQName* in the MQPMO structure with the name of the local queue to which the message is put, and *ResolvedQMgrName* with the name of the local queue manager that hosts the local queue. For more information about MQPMO_RESOLVE_LOCAL_Q, see topic [MQOO_RESOLVE_LOCAL_Q](#).

If you are authorized to put to a queue, you have the required authority to specify this flag on the MQPUT call; no special authority is needed.

Default option. If you need none of the options described, use the following option:

MQPMO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values. MQPMO_NONE is defined to aid program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

MQPMO_NONE is an input field. The initial value of the *Options* field is MQPMO_NONE.

OriginalMsgHandle (MQHMSG)

This is an optional handle to a message. It might have been previously retrieved from a queue. The use of this handle is subject to the value of the *Action* field; see also [NewMsgHandle](#).

The contents of the original message handle will not be altered by the MQPUT or MQPUT1 call.

This is an input field. The initial value of this field is **MQHM_NONE**. This field is ignored if Version is less than **MQPMO_VERSION_3**.

PubLevel (MQLONG)

The initial value of this field is 9. The level of subscription targeted by this publication. Only those subscriptions with the highest SubLevel less than or equal to this value receive this publication. This value must be in the range zero to 9; zero is the lowest level. However, if a publication has been retained, it is no longer available to subscribers at higher levels because it is republished at PubLevel 1.

For information, see [Intercepting publications](#).

PutMsgRecFields (MQLONG)

This field contains flags that indicate which MQPMR fields are present in the put message records provided by the application. Use *PutMsgRecFields* only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero, or both *PutMsgRecOffset* and *PutMsgRecPtr* are zero.

For fields that are present, the queue manager uses for each destination the values from the fields in the corresponding put message record. For fields that are absent, the queue manager uses the values from the MQMD structure.

Use one or more of the following flags to indicate which fields are present in the put message records:

MQPMRF_MSG_ID

Message-identifier field is present.

MQPMRF_CORREL_ID

Correlation-identifier field is present.

MQPMRF_GROUP_ID

Group-identifier field is present.

MQPMRF_FEEDBACK

Feedback field is present.

MQPMRF_ACCOUNTING_TOKEN

Accounting-token field is present.

If you specify this flag, specify either MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT in the *Options* field; if this condition is not satisfied, the call fails with reason code MQRC_PMO_RECORD_FLAGS_ERROR.

If no MQPMR fields are present, the following can be specified:

MQPMRF_NONE

No put-message record fields are present.

If this value is specified, either *RecsPresent* must be zero, or both *PutMsgRecOffset* and *PutMsgRecPtr* must be zero.

MQPMRF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

If *PutMsgRecFields* contains flags that are not valid, or put message records are provided but *PutMsgRecFields* has the value MQPMRF_NONE, the call fails with reason code MQRC_PMO_RECORD_FLAGS_ERROR.

This is an input field. The initial value of this field is MQPMRF_NONE. This field is ignored if *Version* is less than MQPMO_VERSION_2.

PutMsgRecOffset (MQLONG)

This is the offset in bytes of the first MQPMR put message record from the start of the MQPMO structure. The offset can be positive or negative. *PutMsgRecOffset* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

When the message is being put to a distribution list, an array of one or more MQPMR put message records can be provided in order to specify certain properties of the message for each destination individually; these properties are:

- Message identifier
- Correlation identifier
- Group identifier
- Feedback value
- Accounting token

You do not need to specify all these properties, but whatever subset you choose, specify the fields in the correct order. See the description of the MQPMR structure for further details.

Usually, there must be as many put message records as there are object records specified by MQOD when the distribution list is opened; each put message record supplies the message properties for the queue identified by the corresponding object record. Queues in the distribution list that fail to open must still have put message records allocated for them at the appropriate positions in the array, although the message properties are ignored in this case.

The number of put message records can differ from the number of object records. If there are fewer put message records than object records, the message properties for the destinations that do not have put message records are taken from the corresponding fields in the message descriptor MQMD. If there are more put message records than object records, the excess are not used (although it must still be possible to access them). Put message records are optional, but if they are supplied there must be *RecsPresent* of them.

Provide the put message records in a similar way to the object records in MQOD, either by specifying an offset in *PutMsgRecOffset*, or by specifying an address in *PutMsgRecPtr*; for details of how to do this, see the *ObjectRecOffset* field described in “MQOD - Object descriptor” on page 455.

No more than one of *PutMsgRecOffset* and *PutMsgRecPtr* can be used; the call fails with reason code MQRC_PUT_MSG_RECORDS_ERROR if both are nonzero.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQPMO_VERSION_2.

PutMsgRecPtr (MQPTR)

This is the address of the first MQPMR put message record. Use *PutMsgRecPtr* only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

You can use either *PutMsgRecPtr* or *PutMsgRecOffset* can be used to specify the put message records, but not both; for details, see “PutMsgRecOffset (MQLONG)” on page 488. If you do not use *PutMsgRecPtr*, set it to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQPMO_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

RecsPresent (MQLONG)

This is the number of MQPMR put message records or MQRR response records that have been provided by the application. This number can be greater than zero only if the message is being put to a distribution list. Put message records and response records are optional; the application need not provide any records, or it can choose to provide records of only one type. However, if the application provides records of both types, it must provide *RecsPresent* records of each type.

The value of *RecsPresent* need not be the same as the number of destinations in the distribution list. If too many records are provided, the excess are not used; if too few records are provided, default values are used for the message properties for those destinations that do not have put message records (see *PutMsgRecOffset*).

If *RecsPresent* is less than zero, or is greater than zero but the message is not being put to a distribution list, the call fails with reason code MQRC_RECS_PRESENT_ERROR.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQPMO_VERSION_2.

ResolvedQMgrName (MQCHAR48)

This is the name of the destination queue manager after name resolution has been performed by the local queue manager. The name returned is the name of the queue manager that owns the queue identified by *ResolvedQName*, and can be the name of the local queue manager.

If *ResolvedQName* is a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *ResolvedQMgrName* is the name of the queue-sharing group. If the queue is owned by some other queue-sharing group, *ResolvedQName* can be the name of the queue-sharing group or the name of a queue manager that is a member of the queue-sharing group (the nature of the value returned is determined by the queue definitions that exist at the local queue manager).

A nonblank value is returned only if the object is a single queue; if the object is a distribution list or a topic, the value returned is undefined.

This is an output field. The length of this field is given by `MQ_Q_MGR_NAME_LENGTH`. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ResolvedQName (MQCHAR48)

This is the name of the destination queue after name resolution has been performed by the local queue manager. The name returned is the name of a queue that exists on the queue manager identified by *ResolvedQMgrName*.

A nonblank value is returned only if the object is a single queue; if the object is a distribution list or a topic, the value returned is undefined.

This is an output field. The length of this field is given by `MQ_Q_NAME_LENGTH`. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ResponseRecOffset (MQLONG)

This is the offset in bytes of the first MQRR response record from the start of the MQPMO structure. The offset can be positive or negative. *ResponseRecOffset* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

When putting the message to a distribution list, you can provide an array of one or more MQRR response records to identify the queues to which the message was not sent successfully (*CompCode* field in MQRR), and the reason for each failure (*Reason* field in MQRR). The message might not have been sent either because the queue failed to open, or because the put operation failed. The queue manager sets the response records only when the outcome of the call is mixed (that is, some messages were sent successfully while others failed, or all failed but for differing reasons); reason code `MQRC_MULTIPLE_REASONS` from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *Reason* parameter of the MQPUT or MQPUT1 call, and the response records are not set.

Usually, there are as many response records as there are object records specified by MQOD when the distribution list is opened; when necessary, each response record is set to the completion code and reason code for the put to the queue identified by the corresponding object record. Queues in the distribution list that fail to open must still have response records allocated for them at the appropriate positions in the array, although they are set to the completion code and reason code resulting from the open operation, rather than the put operation.

The number of response records can differ from the number of object records. If there are fewer response records than object records, the application might not be able to identify all the destinations for which the put operation failed, or the reasons for the failures. If there are more response records than object records, the excess are not used (although it must still be possible to access them). Response records are optional, but if they are supplied there must be *RecsPresent* of them.

Provide the response records in a similar way to the object records in MQOD, either by specifying an offset in *ResponseRecOffset*, or by specifying an address in *ResponseRecPtr*; for details of how to do this, see the *ObjectRecOffset* field described in “MQOD - Object descriptor” on page 455. However, use no more than one of *ResponseRecOffset* and *ResponseRecPtr*; the call fails with reason code `MQRC_RESPONSE_RECORDS_ERROR` if both are nonzero.

For the MQPUT1 call, this field must be zero. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is 0. This field is ignored if *Version* is less than MQPMO_VERSION_2.

ResponseRecPtr (MQPTR)

This is the address of the first MQRR response record. *ResponseRecPtr* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

Use either *ResponseRecPtr* or *ResponseRecOffset* to specify the response records, but not both; for details, see [“ResponseRecOffset \(MQLONG\)” on page 490](#). If you do not use *ResponseRecPtr* set it to the null pointer or null bytes.

For the MQPUT1 call, this field must be the null pointer or null bytes. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is ignored if *Version* is less than MQPMO_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQPMO_STRUC_ID

Identifier for put-message options structure.

For the C programming language, the constant MQPMO_STRUC_ID_ARRAY is also defined; this has the same value as MQPMO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQPMO_STRUC_ID.

Timeout (MQLONG)

This is a reserved field; its value is not significant. The initial value of this field is -1.

UnknownDestCount (MQLONG)

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that resolve to remote queues. Messages that the queue manager retains temporarily in distribution-list form count as the number of individual destinations that those distribution lists contain. This field is also set when putting a message to a single queue that is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_1.

This field is undefined on z/OS because distribution lists are not supported.

Version (MQLONG)

Structure version number.

The value must be one of the following:

MQPMO_VERSION_1

Version-1 put-message options structure.

This version is supported in all environments.

MQPMO_VERSION_2

Version-2 put-message options structure.

This version is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

MQPMO_VERSION_3

Version-3 put-message options structure.

This version is supported in all environments.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQPMO_CURRENT_VERSION

Current version of put-message options structure.

This is always an input field. The initial value of this field is MQPMO_VERSION_1.

Initial values and language declarations for MQPMO

<i>Table 517. Initial values of fields in MQPMO</i>		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQPMO_STRUC_ID	'PMO'
<i>Version</i>	MQPMO_VERSION_1	1
<i>Options</i>	MQPMO_NONE	0
<i>Timeout</i>	None	-1
<i>Context</i>	None	0
<i>KnownDestCount</i>	None	0
<i>UnknownDestCount</i>	None	0
<i>InvalidDestCount</i>	None	0
<i>ResolvedQName</i>	None	Null string or blanks
<i>ResolvedQMGrName</i>	None	Null string or blanks
<i>RecsPresent</i>	None	0
<i>PutMsgRecFields</i>	MQPMRF_NONE	0
<i>PutMsgRecOffset</i>	None	0
<i>ResponseRecOffset</i>	None	0
<i>PutMsgRecPtr</i>	None	Null pointer or null bytes
<i>ResponseRecPtr</i>	None	Null pointer or null bytes
<i>OriginalMsgHandle</i>	MQHM_NONE	0
<i>NewMsgHandle</i>	MQHM_NONE	0
<i>Action</i>	MQACTP_NEW	0
<i>PubLevel</i>	None	9

Table 517. Initial values of fields in MQPMO (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQPMO_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQPMO MyPMO = {MQPMO_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQPMO MQPMO;
struct tagMQPMO {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;           /* Structure version number */
    MQLONG     Options;           /* Options that control the action of
                                   MQPUT and MQPUT1 */
    MQLONG     Timeout;           /* Reserved */
    MQHOBJ     Context;           /* Object handle of input queue */
    MQLONG     KnownDestCount;    /* Number of messages sent
                                   successfully to local queues */
    MQLONG     UnknownDestCount;  /* Number of messages sent
                                   successfully to remote queues */
    MQLONG     InvalidDestCount;  /* Number of messages that could not
                                   be sent */
    MQCHAR48   ResolvedQName;     /* Resolved name of destination
                                   queue */
    MQCHAR48   ResolvedQMgrName;  /* Resolved name of destination queue
                                   manager */
    /* Ver:1 */
    MQLONG     RecsPresent;        /* Number of put message records or
                                   response records present */
    MQLONG     PutMsgRecFields;    /* Flags indicating which MQPMR fields
                                   are present */
    MQLONG     PutMsgRecOffset;    /* Offset of first put message record
                                   from start of MQPMO */
    MQLONG     ResponseRecOffset;  /* Offset of first response record
                                   from start of MQPMO */
    MQPTR      PutMsgRecPtr;       /* Address of first put message
                                   record */
    MQPTR      ResponseRecPtr;     /* Address of first response record */
    /* Ver:2 */
    MQHMSG     OriginalMsgHandle;  /* Original message handle */
    MQHMSG     NewMsgHandle;       /* New message handle */
    MQLONG     Action;             /* The action being performed */
    MQLONG     PubLevel;           /* Subscription level */
    /* Ver:3 */
};
```

COBOL declaration

```
** MQPMO structure
10 MQPMO.
** Structure identifier
15 MQPMO-STRUCID PIC X(4).
** Structure version number
15 MQPMO-VERSION PIC S9(9) BINARY.
** Options that control the action of MQPUT and MQPUT1
15 MQPMO-OPTIONS PIC S9(9) BINARY.
** Reserved
15 MQPMO-TIMEOUT PIC S9(9) BINARY.
** Object handle of input queue
15 MQPMO-CONTEXT PIC S9(9) BINARY.
** Number of messages sent successfully to local queues
15 MQPMO-KNOWNDESTCOUNT PIC S9(9) BINARY.
** Number of messages sent successfully to remote queues
15 MQPMO-UNKNOWNDESTCOUNT PIC S9(9) BINARY.
** Number of messages that could not be sent
```

```

15 MQPMO-INVALIDDESTCOUNT PIC S9(9) BINARY.
** Resolved name of destination queue
15 MQPMO-RESOLVEDQNAME PIC X(48).
** Resolved name of destination queue manager
15 MQPMO-RESOLVEDQMGRNAME PIC X(48).
** Number of put message records or response records present
15 MQPMO-RECSPRESENT PIC S9(9) BINARY.
** Flags indicating which MQPMR fields are present
15 MQPMO-PUTMSGRECFIELDS PIC S9(9) BINARY.
** Offset of first put message record from start of MQPMO
15 MQPMO-PUTMSGRECOFFSET PIC S9(9) BINARY.
** Offset of first response record from start of MQPMO
15 MQPMO-RESPONSERECOFFSET PIC S9(9) BINARY.
** Address of first put message record
15 MQPMO-PUTMSGRECPtr POINTER.
** Address of first response record
15 MQPMO-RESPONSERECPtr POINTER.
** Original message handle
15 MQPMO-ORIGINALMSGHANDLE PIC S9(18) BINARY.
** New message handle
15 MQPMO-NEWMSGHANDLE PIC S9(18) BINARY.
** The action being performed
15 MQPMO-ACTION PIC S9(9) BINARY.
** Publish level
15 MQPMO-PUBLEVEL PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
  1 MQPMO based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version number */
    3 Options          fixed bin(31),    /* Options that control the action
                                     of MQPUT and MQPUT1 */

    3 Timeout          fixed bin(31),    /* Reserved */
    3 Context          fixed bin(31),    /* Object handle of input queue */
    3 KnownDestCount   fixed bin(31),    /* Number of messages sent
                                     successfully to local queues */
    3 UnknownDestCount fixed bin(31),    /* Number of messages sent
                                     successfully to remote queues */
    3 InvalidDestCount fixed bin(31),    /* Number of messages that could
                                     not be sent */
    3 ResolvedQName     char(48),        /* Resolved name of destination
                                     queue */
    3 ResolvedQMgrName  char(48),        /* Resolved name of destination
                                     queue manager */
    3 RecsPresent       fixed bin(31),    /* Number of put message records or
                                     response records present */
    3 PutMsgRecFields   fixed bin(31),    /* Flags indicating which MQPMR
                                     fields are present */
    3 PutMsgRecOffset   fixed bin(31),    /* Offset of first put message
                                     record from start of MQPMO */
    3 ResponseRecOffset fixed bin(31),    /* Offset of first response record
                                     from start of MQPMO */
    3 PutMsgRecPtr      pointer,          /* Address of first put message
                                     record */
    3 ResponseRecPtr    pointer,          /* Address of first response
                                     record */
    3 OriginalMsgHandle fixed bin(63),    /* Original message handle */
    3 NewMsgHandle       fixed bin(63);   /* New message handle */
    3 Action            fixed bin(31);    /* The action being performed */
    3 PubLevel          fixed bin(31);    /* Publish level */

```

High Level Assembler declaration

MQPMO	DSECT	
MQPMO_STRUCID	DS	CL4 Structure identifier
MQPMO_VERSION	DS	F Structure version number
MQPMO_OPTIONS	DS	F Options that control the action of
*		MQPUT and MQPUT1
MQPMO_TIMEOUT	DS	F Reserved
MQPMO_CONTEXT	DS	F Object handle of input queue
MQPMO_KNOWNDESTCOUNT	DS	F Number of messages sent successfully
*		to local queues
MQPMO_UNKNOWNDESTCOUNT	DS	F Number of messages sent successfully
*		to remote queues
MQPMO_INVALIDDESTCOUNT	DS	F Number of messages that could not be
*		sent

MQPMO_RESOLVEDQNAME	DS	CL48	Resolved name of destination queue
MQPMO_RESOLVEDQMGRNAME	DS	CL48	Resolved name of destination queue manager
*MQPMO_RECSPRESENT	DS	F	Number of put message records or response records present
*MQPMO_PUTMSGRECFIELDS	DS	F	Flags indicating which MQPMR fields are present
*MQPMO_PUTMSGRECOFFSET	DS	F	Offset of first put message record from start of MQPMO
*MQPMO_RESPONSERECOFFSET	DS	F	Offset of first response record from start of MQPMO
*MQPMO_PUTMSGRECPtr	DS	F	Address of first put message record
*MQPMO_RESPONSERECPtr	DS	F	Address of first response record
MQPMO_ORIGINALMSGHANDLE	DS	D	Original message handle
MQPMO_NEWMSGHANDLE	DS	D	New message handle
MQPMO_ACTION	DS	F	The action being performed
MQPMO_PUBLEVEL	DS	F	Publish level
*MQPMO_LENGTH	EQU	*-MQPMO	
	ORG	MQPMO	
MQPMO_AREA	DS	CL(MQPMO_LENGTH)	

Visual Basic declaration

```

Type MQPMO
    StructId      As String*4  'Structure identifier'
    Version       As Long      'Structure version number'
    Options       As Long      'Options that control the action of'
                                'MQPUT and MQPUT1'
    Timeout       As Long      'Reserved'
    Context       As Long      'Object handle of input queue'
    KnownDestCount As Long      'Number of messages sent successfully'
                                'to local queues'
    UnknownDestCount As Long    'Number of messages sent successfully'
                                'to remote queues'
    InvalidDestCount As Long    'Number of messages that could not be'
                                'sent'
    ResolvedQName As String*48  'Resolved name of destination queue'
    ResolvedQMGrName As String*48 'Resolved name of destination queue'
                                'manager'
    RecsPresent   As Long      'Number of put message records or'
                                'response records present'
    PutMsgRecFields As Long     'Flags indicating which MQPMR fields'
                                'are present'
    PutMsgRecOffset As Long     'Offset of first put message record'
                                'from start of MQPMO'
    ResponseRecOffset As Long    'Offset of first response record from'
                                'start of MQPMO'
    PutMsgRecPtr   As MQPTR     'Address of first put message record'
    ResponseRecPtr As MQPTR     'Address of first response record'
End Type

```

MQPMR - Put-message record

The following table summarizes the fields in the structure.

Table 518. Fields in MQPMR		
Field	Description	Topic
<i>MsgId</i>	Message identifier	MsgId
<i>CorrelId</i>	Correlation identifier	CorrelId
<i>GroupId</i>	Group identifier	GroupId
<i>Feedback</i>	Feedback or reason code	Feedback
<i>AccountingToken</i>	Accounting token	AccountingToken

Overview for MQPMR

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

Purpose: Use the MQPMR structure to specify various message properties for a single destination when putting a message to a distribution list. MQPMR is an input/output structure for the MQPUT and MQPUT1 calls.

Character set and encoding: Data in MQPMR must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQPUT or MQPUT1 call, you can specify different values for each destination queue in a distribution list. Some of the fields are input only, others are input/output.

Note: This structure is unusual in that it does not have a fixed layout. The fields in this structure are optional, and the presence or absence of each field is indicated by the flags in the *PutMsgRecFields* field in MQPMO. Fields that are present **must occur in the following order** :

- *MsgId*
- *CorrelId*
- *GroupId*
- *Feedback*
- *AccountingToken*

Fields that are absent occupy no space in the record.

Because MQPMR does not have a fixed layout, no definition of it is provided in the header, COPY, and INCLUDE files for the supported programming languages. The application programmer must create a declaration containing the fields that are required by the application, and set the flags in *PutMsgRecFields* to indicate the fields that are present.

Fields for MQPMR

The MQPMR structure contains the following fields; the fields are described in **alphabetical order**:

AccountingToken (MQBYTE32)

This is the accounting token to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *AccountingToken* field in MQMD for a put to a single queue. See the description of *AccountingToken* in [“MQMD - Message descriptor” on page 396](#) for information about the content of this field.

If this field is not present, the value in MQMD is used.

This is an input field.

CorrelId (MQBYTE24)

This is the correlation identifier to be used for the message sent to the queue with a name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *CorrelId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *CorrelId* field.

If MQPMO_NEW_CORREL_ID is specified, a *single* new correlation identifier is generated and used for all the destinations in the distribution list, regardless of whether they have MQPMR records. This is different from the way that MQPMO_NEW_MSG_ID is processed (see *MsgId* field).

This is an input/output field.

Feedback (MQLONG)

This is the feedback code to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *Feedback* field in MQMD for a put to a single queue.

If this field is not present, the value in MQMD is used.

This is an input field.

GroupId (MQBYTE24)

GroupId is the group identifier to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *GroupId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *GroupId* field. The value is processed as documented in [Physical order on a queue](#), but with the following differences:

- GroupId is created from the QMName and a timestamp. Therefore to keep a GroupId unique keep queue manager names unique too. Also do not set the clocks back on the queue managers machine.
- In those cases where a new group identifier would be used, the queue manager generates a different group identifier for each destination (that is, no two destinations have the same group identifier).
- In those cases where the value in the field would be used, the call fails with reason code MQRC_GROUP_ID_ERROR

This is an input/output field.

MsgId (MQBYTE24)

This is the message identifier to be used for the message sent to the queue with a name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MsgId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *MsgId* field. If that value is MQMI_NONE, a new message identifier is generated for *each* of those destinations (that is, no two of those destinations have the same message identifier).

If MQPMO_NEW_MSG_ID is specified, new message identifiers are generated for all the destinations in the distribution list, regardless of whether they have MQPMR records. This is different from the way that MQPMO_NEW_CORREL_ID is processed (see *CorrelId* field).

This is an input/output field.

Initial values and language declarations for MQPMR

There are no initial values defined for this structure, as no structure declarations are provided in the header, COPY, and INCLUDE files for the supported programming languages. The sample declarations show how to declare the structure if all the fields are required.

C declaration

```
typedef struct tagMQPMR MQPMR;
struct tagMQPMR {
    MQBYTE24  MsgId;           /* Message identifier */
    MQBYTE24  CorrelId;        /* Correlation identifier */
    MQBYTE24  GroupId;        /* Group identifier */
    MQLONG    Feedback;       /* Feedback or reason code */
    MQBYTE32  AccountingToken; /* Accounting token */
};
```

COBOL declaration

```
** MQPMR structure
10 MQPMR.
**   Message identifier
15 MQPMR-MSGID          PIC X(24).
**   Correlation identifier
15 MQPMR-CORRELID       PIC X(24).
**   Group identifier
15 MQPMR-GROUPID        PIC X(24).
**   Feedback or reason code
15 MQPMR-FEEDBACK       PIC S9(9) BINARY.
**   Accounting token
15 MQPMR-ACCOUNTINGTOKEN PIC X(32).
```

PL/I declaration

```
dcl
1 MQPMR based,
3 MsgId          char(24),      /* Message identifier */
3 CorrelId        char(24),      /* Correlation identifier */
3 GroupId         char(24),      /* Group identifier */
3 Feedback        fixed bin(31), /* Feedback or reason code */
3 AccountingToken char(32);      /* Accounting token */
```

Visual Basic declaration

```
Type MQPMR
MsgId          As MQBYTE24 'Message identifier'
CorrelId        As MQBYTE24 'Correlation identifier'
GroupId         As MQBYTE24 'Group identifier'
Feedback        As Long     'Feedback or reason code'
AccountingToken As MQBYTE32 'Accounting token'
End Type
```

MQRFH - Rules and formatting header

This section describes the rules and formatting header, what fields it contains, and initial values of those fields.

Overview for MQRFH

Availability: All IBM MQ systems, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQRFH structure defines the layout of the rules and formatting header. Use this header to send string data in the form of name/value pairs.

Format name: MQFMT_RF_HEADER.

Character set and encoding: The fields in the MQRFH structure (including *NameValueString*) are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes the MQRFH, or by those fields in the MQMD structure if the MQRFH is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

Fields for MQRFH

The MQRFH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This specifies the character set identifier of the data that follows *NameValueString*; it does not apply to character data in the MQRFH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

MQCCSI_INHERIT

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

MQCCSI_INHERIT cannot be used if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

The initial value of this field is MQCCSI_UNDEFINED.

Encoding (MQLONG)

This specifies the numeric encoding of the data that follows *NameValueString*; it does not apply to numeric data in the MQRFH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is MQENC_NATIVE.

Flags (MQLONG)

The following can be specified:

MQRFH_NONE

No flags.

The initial value of this field is MQRFH_NONE.

Format (MQCHAR8)

This specifies the format name of the data that follows *NameValueString*.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *Format* field in MQMD.

The initial value of this field is MQFMT_NONE.

NameValueString (MQCHARn)

This is a variable-length character string containing name/value pairs in the form:

```
name1 value1 name2 value2 name3 value3 ...
```

Each name or value must be separated from the adjacent name or value by one or more blank characters; these blanks are not significant. A name or value can contain significant blanks by prefixing and suffixing the name or value with double quotation marks; all characters between the open double quotation mark and the matching closing double quotation mark are treated as significant. In the following example, the name is FAMOUS_WORDS, and the value is Hello World:

```
FAMOUS_WORDS "Hello World"
```

A name or value can contain any characters other than the null character (which acts as a delimiter for *NameValueString*). However, to assist interoperability an application can restrict names to the following characters:

- First character: upper or lowercase alphabetic (A through Z, or a through z), or underscore.
- Subsequent characters: upper or lowercase alphabetic, decimal digit (0 through 9), underscore, hyphen, or dot.

If a name or value contains one or more double quotation marks, the name or value must be enclosed in double quotation marks, and each double quotation mark within the string must be doubled:

```
Famous_Words "The program displayed ""Hello World"""
```

Names and values are case sensitive, that is, lowercase letters are not considered to be the same as uppercase letters. For example, FAMOUS_WORDS and Famous_Words are two different names.

The length in bytes of *NameValueString* is equal to *StrucLength* minus MQRFH_STRUC_LENGTH_FIXED. To avoid problems converting the user data in some environments, make this length a multiple of four. Pad *NameValueString* with blanks to this length, or terminate it earlier by placing a null character following the last significant character in the string. The null character and the bytes following it, up to the specified length of *NameValueString*, are ignored.

Note: Because the length of this field is not fixed, the field is omitted from the declarations of the structure that are provided for the supported programming languages.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQRFH_STRUC_ID

Identifier for rules and formatting header structure.

For the C programming language, the constant MQRFH_STRUC_ID_ARRAY is also defined; this has the same value as MQRFH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQRFH_STRUC_ID.

StrucLength (MQLONG)

This is the length in bytes of the MQRFH structure, including the *NameValueString* field at the end of the structure. The length does *not* include any user data that follows the *NameValueString* field.

To avoid problems converting the user data in some environments, *StrucLength* must be a multiple of four.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *NameValueString* field:

MQRFH_STRUC_LENGTH_FIXED

Length of fixed part of MQRFH structure.

The initial value of this field is MQRFH_STRUC_LENGTH_FIXED.

Version (MQLONG)

This is the structure version number; the value must be:

MQRFH_VERSION_1

Version-1 rules and formatting header structure.

The initial value of this field is MQRFH_VERSION_1.

Initial values and language declarations for MQRFH

Table 519. Initial values of fields in MQRFH for MQRFH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQRFH_STRUC_ID	'RFH~'
<i>Version</i>	MQRFH_VERSION_1	1
<i>StrucLength</i>	MQRFH_STRUC_LENGTH_FIXED	32
<i>Encoding</i>	MQENC_NATIVE	Depends on environment
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQRFH_NONE	0

Table 519. Initial values of fields in MQRFH for MQRFH (continued)

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. In the C programming language, the macro variable MQRFH_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre> MQRFH MyRFH = {MQRFH_DEFAULT}; </pre> 		

C declaration

```

typedef struct tagMQRFH MQRFH;
struct tagMQRFH {
    MQCHAR4  StrucId;           /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   StrucLength;      /* Total length of MQRFH including
                               NameValueString */
    MQLONG   Encoding;         /* Numeric encoding of data that follows
                               NameValueString */
    MQLONG   CodedCharSetId;   /* Character set identifier of data that
                               follows NameValueString */
    MQCHAR8  Format;           /* Format name of data that follows
                               NameValueString */
    MQLONG   Flags;            /* Flags */
};

```

COBOL declaration

```

**  MQRFH structure
10 MQRFH.
**  Structure identifier
15 MQRFH-STRUCID      PIC X(4).
**  Structure version number
15 MQRFH-VERSION     PIC S9(9) BINARY.
**  Total length of MQRFH including NAMEVALUESTRING
15 MQRFH-STRUCLength PIC S9(9) BINARY.
**  Numeric encoding of data that follows NAMEVALUESTRING
15 MQRFH-ENCODING    PIC S9(9) BINARY.
**  Character set identifier of data that follows NAMEVALUESTRING
15 MQRFH-CODEDCHARSETID PIC S9(9) BINARY.
**  Format name of data that follows NAMEVALUESTRING
15 MQRFH-FORMAT      PIC X(8).
**  Flags
15 MQRFH-FLAGS       PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQRFH based,
3 StrucId      char(4),          /* Structure identifier */
3 Version      fixed bin(31),    /* Structure version number */
3 StrucLength  fixed bin(31),    /* Total length of MQRFH including
                               NameValueString */
3 Encoding     fixed bin(31),    /* Numeric encoding of data that
                               follows NameValueString */
3 CodedCharSetId fixed bin(31),  /* Character set identifier of data
                               that follows NameValueString */
3 Format        char(8),          /* Format name of data that follows
                               NameValueString */
3 Flags        fixed bin(31);    /* Flags */

```

High Level Assembler declaration

```

MQRFH          DSECT
MQRFH_STRUCID  DS   CL4  Structure identifier
MQRFH_VERSION  DS   F    Structure version number

```

MQRFH_STRUCLength	DS	F	Total length of MQRFH including
* MQRFH_ENCODING	DS	F	NAMEVALUESTRING
* MQRFH_CODEDCHARSETID	DS	F	Numeric encoding of data that follows
* MQRFH_FORMAT	DS	CL8	NAMEVALUESTRING
* MQRFH_FLAGS	DS	F	Character set identifier of data that follows
* MQRFH_LENGTH	EQU	*-MQRFH	NAMEVALUESTRING
MQRFH_AREA	ORG	MQRFH	Flags
	DS	CL(MQRFH_LENGTH)	

Visual Basic declaration

```

Type MQRFH
  StrucId      As String*4 'Structure identifier'
  Version      As Long     'Structure version number'
  StrucLength  As Long     'Total length of MQRFH including'
                          'NameValueString'
  Encoding     As Long     'Numeric encoding of data that follows'
                          'NameValueString'
  CodedCharSetId As Long   'Character set identifier of data that'
                          'follows NameValueString'
  Format       As String*8 'Format name of data that follows'
                          'NameValueString'
  Flags       As Long     'Flags'
End Type

```

MQRFH2 - Rules and formatting header 2

This section describes the rules and formatting header 2, what fields it contains, and initial values of those fields.

Overview for MQRFH2

Availability

All IBM MQ systems, plus IBM MQ MQI clients connected to these systems.

Purpose

The MQRFH2 header is based on the MQRFH header, but it allows Unicode strings to be transported without translation, and it can carry numeric data types.

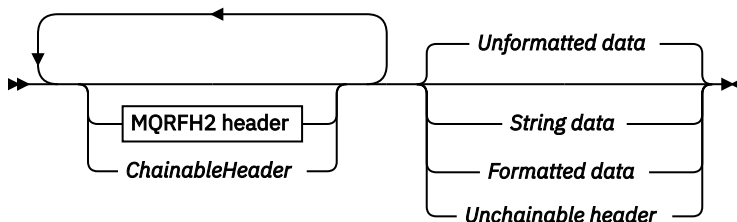
The MQRFH2 structure defines the format of the version-2 rules and formatting header. Use this header to send data that has been encoded using an XML-like syntax. A message can contain two or more MQRFH2 structures in series, with user data optionally following the last MQRFH2 structure in the series.

Format name

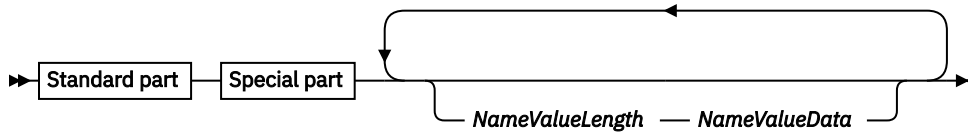
MQFMT_RF_HEADER_2

Syntax

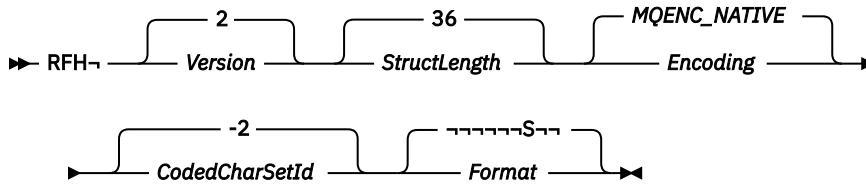
IBM MQ Message



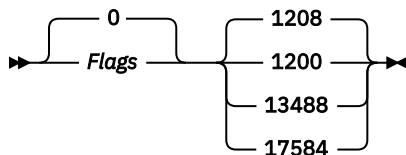
MQRFH2 header



Standard part



Special part



Character set and encoding

Special rules apply to the character set and encoding used for the MQRFH2 structure:

- Fields other than *NameValueData* are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes MQRFH2, or by those fields in the MQMD structure if the MQRFH2 is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

When MQGMO_CONVERT is specified on the MQGET call, the queue manager converts the MQRFH2 fields, other than *NameValueData*, to the requested character set and encoding.

- NameValueData* is in the character set given by the *NameValueCCSID* field. Only the listed Unicode character sets are valid for *NameValueCCSID*; see the description of *NameValueCCSID* for details.

Some character sets have a representation that depends on the encoding. If *NameValueCCSID* is one of these character sets, *NameValueData* must be in the same encoding as the other fields in the MQRFH2.

When MQGMO_CONVERT is specified on the MQGET call, the queue manager converts *NameValueData* to the requested encoding, but does not change its character set.

Fields for MQRFH2

The MQRFH2 structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This specifies the character set identifier of the data that follows the last *NameValueData* field; it does not apply to character data in the MQRFH2 structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

MQCCSI_INHERIT

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

MQCCSI_INHERIT cannot be used if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

The initial value of this field is MQCCSI_INHERIT.

Encoding (MQLONG)

This specifies the numeric encoding of the data that follows the last *NameValueData* field; it does not apply to numeric data in the MQRFH2 structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is MQENC_NATIVE.

Flags (MQLONG)

The initial value of this field is MQRFH_NONE. MQRFH_NONE must be specified.

MQRFH_NONE

No flags.

MQRFH_INTERNAL

The MQRFH2 header contains internally set properties.

MQRFH_INTERNAL is for queue manager use.

The top 16 bits, MQRFH_FLAGS_RESTRICTED_MASK, are reserved for flags the queue manager sets. Flags that a user might set are defined in the bottom 16 bits.

Format (MQCHAR8)

This specifies the format name of the data that follows the last *NameValueData* field.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *Format* field in MQMD.

The initial value of this field is MQFMT_NONE.

NameValueCCSID (MQLONG)

This specifies the coded character set identifier of the data in the *NameValueData* field. This is different from the character set of the other strings in the MQRFH2 structure, and can be different from the character set of the data (if any) that follows the last *NameValueData* field at the end of the structure.

NameValueCCSID must have one of the following values:

CCSID	Meaning
1200	UCS-2 open-ended
13488	UCS-2 2.0 subset
17584	UCS-2 2.1 subset (includes the Euro symbol)
1208	UTF-8

For the UCS-2 character sets, the encoding (byte order) of the *NameValueData* must be the same as the encoding of the other fields in the MQRFH2 structure. Surrogate characters (X'D800' through X'DFFF') are not supported.

Note: If *NameValueCCSID* does not have one of the values listed above, and the MQRFH2 structure requires conversion on the MQGET call, the call completes with reason code MQRC_SOURCE_CCSID_ERROR and the message is returned unconverted.

The initial value of this field is 1208.

NameValueData (MQCHARn)

NameValueData is a variable length field that contains a folder containing name/value pairs of message properties. A folder is a variable-length character string containing data encoded using an XML-like syntax. The length in bytes of the character string is given by the *NameValueLength* field that precedes the *NameValueData* field. The length must be a multiple of four.

The *NameValueLength* and *NameValueData* fields are optional, but if present they must occur as a pair and be adjacent. The pair of fields can be repeated as many times as required, for example:

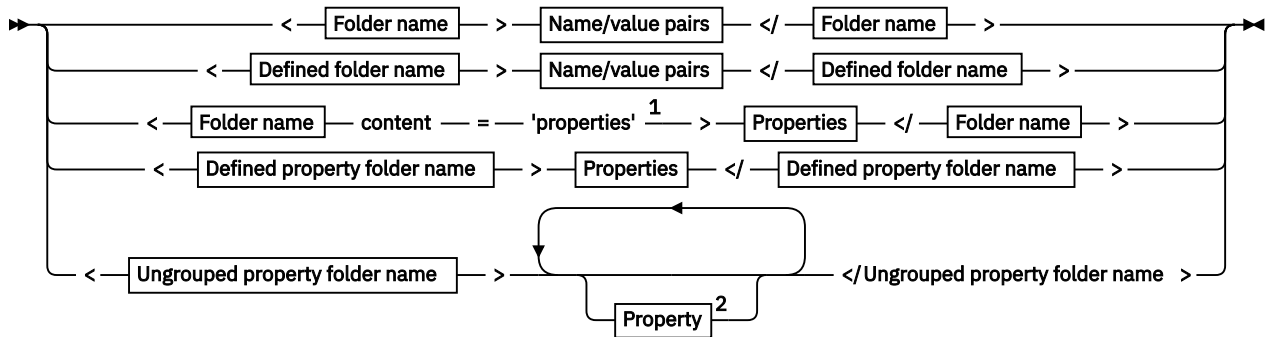
```
length1 data1 length2 data2 length3 data3
```

NameValueData is *not* converted to the character set specified on the MQGET call. Even if the message is retrieved with the MQGMO_CONVERT option in effect *NameValueData* remains in its original character set. However, *NameValueData* is converted to the encoding specified on the MQGET call.

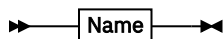
Note:

- Because these fields are optional, they are omitted from the declarations of the structure that are provided for the various programming languages supported.
- The terms "defined" and "reserved" are used in the syntax diagram. "Defined" means that the name is used by IBM MQ. "Reserved" means that the name is reserved for future use by IBM MQ.

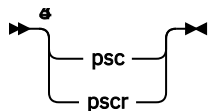
***NameValueData* syntax**



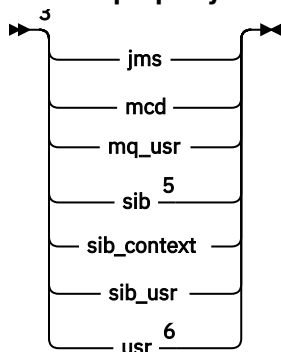
Folder name



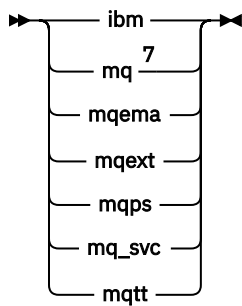
Defined folder name



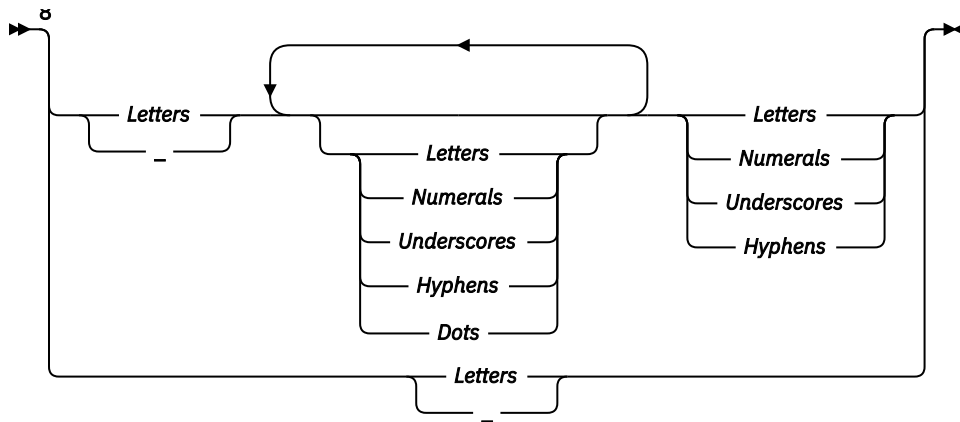
Defined property folder name



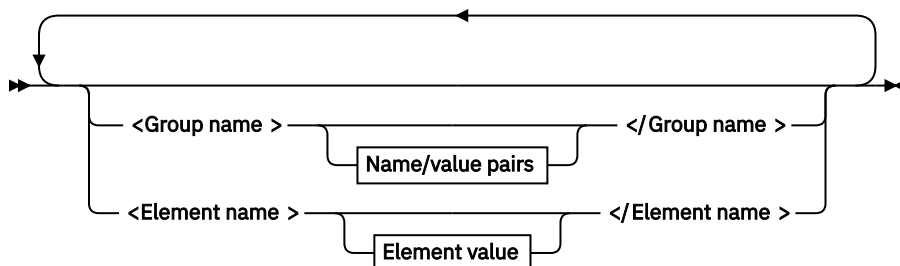
Ungrouped property folder name



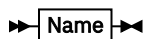
Name



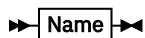
Name/value pairs



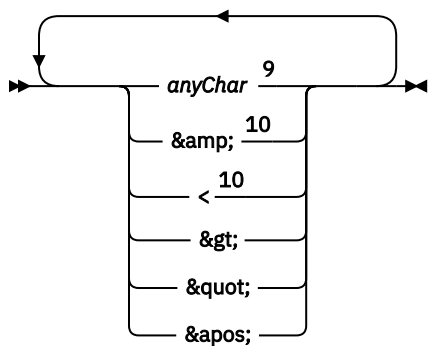
Group name



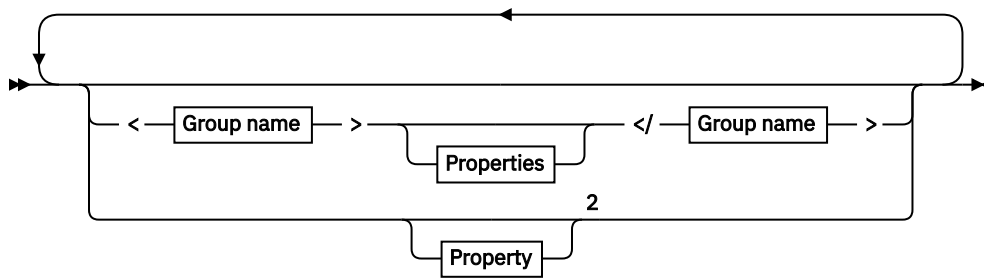
Element name



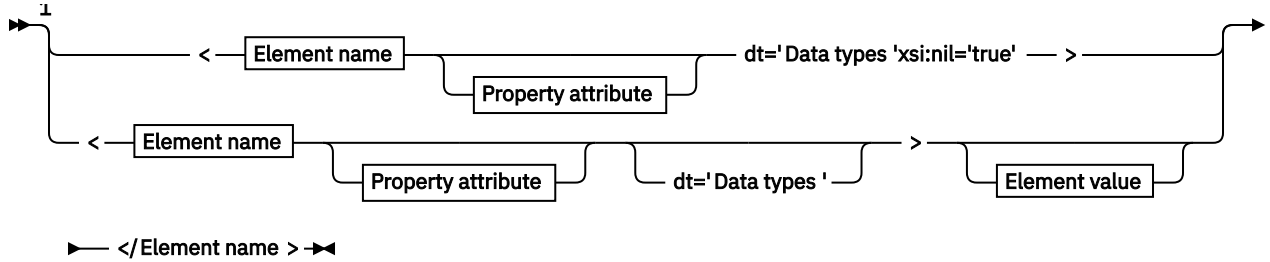
Element value



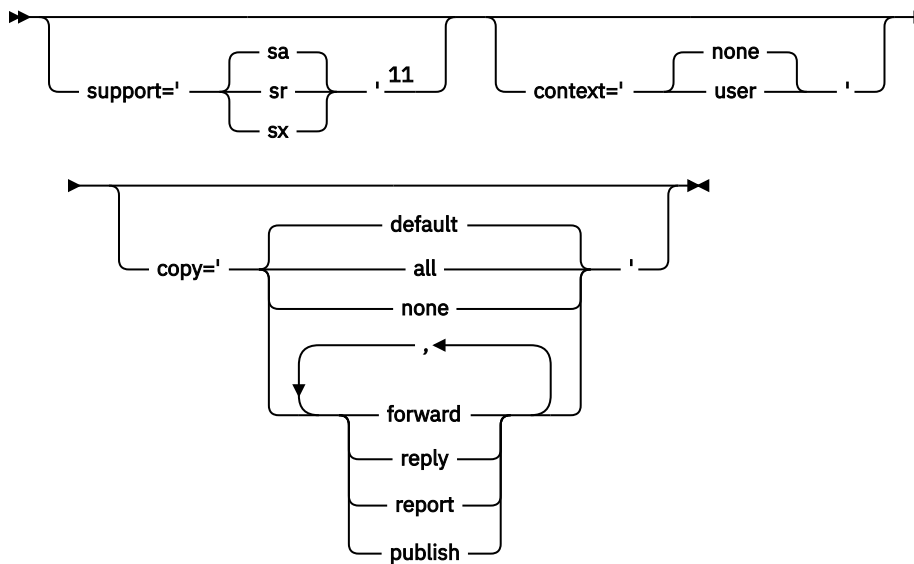
Properties



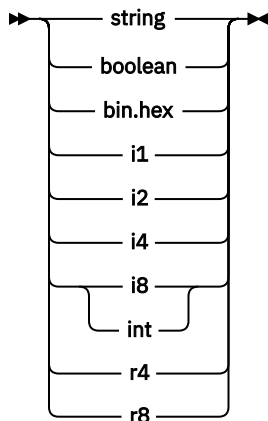
Property



Property attribute



Data types



Notes:

¹ Double quotation marks or single quotation marks are valid.

² Do not use an invalid property name; see [“Invalid property name” on page 518](#). Use a reserved property name only for its defined purpose; see [“Defined property names” on page 518](#).

³ The name must be in lowercase.

⁴ Only one psc and pscx folder is supported.

⁵ Only properties in the first MQRFH2 header are significant. WebSphere Application Server Service Integration Bus ignores sib, sib_context, and sib_usr folders in subsequent MQRFH2 headers.

⁶ Not more than one usr folder must be present in an MQRFH2. Properties in the usr folder must occur no more than once.

⁷ Only properties in the first mq folder are significant. If the folder is UTF-8, only single byte UTF-8 characters are supported. The only white space character is Unicode U+0020.

⁸ Valid characters are defined in the W3C XML specification, and consist essentially of Unicode categories Ll, Lu, Lo, Lt, Nl, Mc, Mn, Lm, and Nd ; see [Unicode character categories](#).

⁹ All characters are significant. Leading and trailing blanks are part of the element value.

¹⁰ Do not use an invalid character; see [“Invalid characters” on page 518](#). Use an escape sequence, rather than these invalid characters.

¹¹ The support property attribute is only valid on the mq folder

Folder name

NameValueData contains a single folder. To create multiple folders, create multiple *NameValueData* fields. You can create multiple *NameValueData* fields in a single MQRFH2 header within a message. Alternatively you can create multiple chained MQRFH2 headers, each containing multiple *NameValueData* fields.

The order of MQRFH2 headers, and the order of *NameValueData* fields makes no difference to the logical contents of a folder. If the same folder is present more than once in a message the folder is parsed as a whole. If the same property occurs in multiple instances of the same folder, it is parsed as a list.

A correct parse of an MQRFH2 is not affected by the alternative ways a folder can be physically stored in a message.

Four folders do not follow this rule. Only the first instance of the mq, sib, sib_context, and sib_usr folder are parsed.

If the same property occurs more than once in the combined contents of the chained MQRFH2 headers, only the first instance of the property is parsed. If a property is set using an API call, such as MQSETMP, and added to an MQRFH2 directly by an application, the API call takes precedence.

A folder name is the name of a folder containing name/value pairs or groups. Groups and name/value pairs can be mixed at the same level in the folder tree; see [Figure 1 on page 508](#). Do not combine a group name and an element name; see [Figure 2 on page 508](#)

```
<group1><nvp1>value</nvp1></group1><group2><nvp2>value</nvp2></group2>
<group3><nvp1> value </nvp1></group3><nvp3> value </nvp3>
```

Figure 1. Correct uses of groups and name/value pairs

```
<group1><nvp1>value</nvp1>value</group1>
```

Figure 2. Incorrect use of groups and name/value pairs

Do not use an invalid or reserved folder name; see [“Invalid path name” on page 518](#) and [“Reserved folder or property folder name” on page 517](#). Use a defined folder name only for its defined purpose; see [“Defined folder name” on page 509](#).

If you add the attribute 'content=properties' to the folder name tag, the folder becomes a property folder; see [Figure 3 on page 509](#).

```
<myFolder></myFolder>
<myPropertyFolder contents='properties'></myPropertyFolder>
```

Figure 3. Example of a folder and a property folder

Folder names are case-sensitive. Folder names and property folder names share the same namespace. They must have different names. Folder1 in [Figure 4 on page 509](#) must be a different name to Folder2 in [Figure 5 on page 509](#).

```
<Folder1><NVP1>value</NVP1></Folder1>
```

Figure 4. Folder1 namespace

```
<Folder2 content='properties'><Property1>value</Property1></Folder2>
```

Figure 5. Folder2 namespace

Groups, properties, and name/value pairs in different folders have different namespaces. Property1 in [Figure 5 on page 509](#) is a different property to Property1 in [Figure 6 on page 509](#).

```
<Folder3 content='properties'><Property1>value</Property1></Folder3>
```

Figure 6. Folder3 namespace

Property folders are different to non-property folders in two important respects:

1. Property folders contain properties, and non-property folders contain name/value pairs. The folders differ slightly, syntactically.
2. Use the defined interfaces, such as the properties MQI, or JMS message properties, to access message properties. The interfaces ensure the property folders in the MQRFH2 are well-formed. A well-formed property folder is interoperable between queue managers on different platforms and different releases.

The message property MQI is a robust way to read and write an MQRFH2, and avoids the difficulties of parsing an MQRFH2 correctly.

Defined folder name

A defined folder name is the name of a folder that is reserved for use by IBM MQ, or another product. Do not create a folder of the same name, and do not add your own name/value pairs to the folders. The defined folders are psc and pscx .

psc and pscx are used by queued publish/subscribe.

A segmented message put with either MQMF_SEGMENT or MQMF_SEGMENTATION_ALLOWED cannot contain an MQRFH2 with a defined folder name. The MQPUT fails with reason code 2443, MQRC_SEGMENTATION_NOT_ALLOWED.

Defined property folder name

A defined property folder name is the name of a property folder that is used by IBM MQ, or another product. For the names of the folders and their contents, see [Property folders](#). Defined property folder

names are a subset of all the folder names reserved by IBM MQ; see [“Reserved folder or property folder name”](#) on page 517.

Any element stored in a defined property folder is a property. An element stored in a defined property folder must not have a content= 'properties' attribute.

You can add properties only to the defined property folders `usr`, `mq_usr`, and `sib_usr`. In other property folders, such as `mq` and `sib`, IBM MQ ignores or throws away properties it does not recognize.

The description of each defined property folder lists the properties that IBM MQ has defined that can be used by application programs. Some of the properties are accessed indirectly by setting or getting a JMS property, and some are accessed directly using the `MQSETMP` and `MQINQMP` MQI calls.

The defined property folders also contain other properties that IBM MQ has reserved, but which applications do not have access to. The names of the reserved properties are not listed. No reserved properties are present in the `usr`, `mq_usr`, and `sib_usr` property folders. But do not create properties with invalid property names; see [“Invalid property name”](#) on page 518.

Property folders

jms

`jms` contains JMS header fields, and JMSX properties that cannot be fully expressed in the MQMD. The `jms` folder is always present in a JMS MQRFH2.

Table 520. <i>jms</i> property name, synonym, data type, and folder			
Property synonym	Property name	Data type	Folder
JMSDestination	<code>jms.Dst</code>	string	<code><jms><Dst>destination</Dst></jms></code>
JMSExpiration	<code>jms.Exp</code>	i8	<code><jms><Exp>expiration</Exp></jms></code>
JMSCorrelation	<code>jms.Cid</code>	string	<code><jms><Cid>correlationId</Cid></jms></code>
JMSDelivery	<code>jms.Dlv</code>	i4	<code><jms><Dlv>delivery</Dlv></jms></code>
JMSPriority	<code>jms.Pri</code>	i4	<code><jms><Pri>priority</Pri></jms></code>
JMSReplyTo	<code>jms.Rto</code>	string	<code><jms><Rto>replyToURI</Rto></jms></code>
JMSTimestamp	<code>jms.Tms</code>	i8	<code><jms><Tms>timestamp</Tms></jms></code>
JMSXGroupID	<code>jms.Gid</code>	string	<code><jms><Gid>groupId</Gid></jms></code>
JMSXGroupSeq	<code>jms.Seq</code>	i4	<code><jms><Seq>messageSequenceNo</Seq></jms></code>

Do not add your own properties in the `jms` folder.

mcd

`mcd` contains properties that describe the format of the message. For example, the message service domain `Msd` property identifies a JMS message as being `JMSTextMessage`, `JMSBytesMessage`, `JMSStreamMessage`, `JMSMapMessage`, `JMSObjectMessage`, or `null`.

The `mcd` folder is always present in a JMS message containing an MQRFH2.

It is always present in a message containing an MQRFH2 sent from IBM Integration Bus. It describes the domain, format, type, and message set of a message.

<i>Table 521. mcd property name, synonym, data type, and folder</i>			
Property synonym	Property name	Data type	Folder
	mcd.Msd	string	<mcd><Msd>messageDomain</Msd></mcd>
	mcd.Set	string	<mcd><Set>messageDomain</Set></mcd>
	mcd.Type	string	<mcd><Type>messageDomain</Type></mcd>
	mcd.Fmt	string	<mcd><Fmt>messageDomain</Fmt></mcd>

Do not add your own properties in the mcd folder.

mq_usr

mq_usr contains application-defined properties that are not exposed as JMS user-defined properties. Properties that do not meet JMS requirements can be placed in this folder.

You can create properties in the mq_usr folder. Properties you create in the mq_usr are like properties you create in new folders with the content='properties' attribute.

sib

sib contains WebSphere Application Server service integration bus (WAS/SIB) system message properties. sib properties are not exposed as JMS properties to IBM MQ JMS applications because they are not of the supported types. For example, some sib properties cannot be exposed as JMS properties because they are byte arrays. Some sib properties are exposed to WAS/SIB applications as JMS_IBM_* properties; these include forward and reverse routing paths properties.

Do not add your own properties in the sib folder.

sib_context

sib_context contains WAS/SIB system message properties that are not exposed to WAS/SIB user applications or as JMS properties. sib_context contains security and transactional properties that are used for web services.

Do not add your own properties in the sib_context folder.

sib_usr

sib_usr contains WAS/SIB user message properties that are not exposed as JMS user properties because they are not of supported types. sib_usr is exposed to WAS/SIB applications in the SIMessage interface. See [Developing Service Integration](#).

The type of a sib_usr property must be bin.hex, and the value must be in the correct format. If an IBM MQ application writes a bin.hex typed element to the folder in the wrong format, the application receives an IOException. If the data type of the property is not bin.hex the application receives a ClassCastException.

Do not attempt to make JMS user properties available to WAS/SIB by using this folder; instead use the usr folder.

You can create properties in the sib_usr folder.

usr

usr contains application-defined JMS properties associated with the message. The usr folder is present only if an application has set an application-defined property.

usr is the default property folder. If a property is set without a folder name, it is placed in the usr folder.

Table 522. <i>usr</i> property name, synonym, data type, and folder.			
The web services property values are described in MQRFH2 SOAP settings			
Property synonym	Property name	Data type	Folder
	usr.contentType	string	<usr><contentType>text/xml; charset=utf-8</contentType></usr>
	usr.endpointURL	string	<usr><endpointURL>URI</endpointURL></usr>
	usr.targetService	string	<usr><targetService>serviceName</targetService></usr>
	usr.soapAction	string	<usr><soapAction>name</soapAction></usr>
	usr.transportVersion	string	<usr><transportVersion>version</transportVersion></usr>

You can create properties in the usr folder.

A segmented message put with either MQMF_SEGMENT or MQMF_SEGMENTATION_ALLOWED cannot contain an MQRFH2 with a defined property folder name. The MQPUT fails with reason code 2443, MQRC_SEGMENTATION_NOT_ALLOWED.

Ungrouped property folder name

ibm

ibm contains properties that are used only by IBM MQ.

Table 523. <i>ibm</i> property name, synonym, data type, and folder			
Property synonym	Property name	Data type	Folder
	ibm.rfp	string	<ibm><rfp>fingerprint</rfp></ibm>

Do not add your own properties in the ibm folder.

mq

mq contains properties that are used only by IBM MQ.

The following restrictions apply to properties in the mq folder:

- Only properties in the first significant mq folder in the message are acted upon by MQ; properties in any other mq folder in the message are ignored.
- Only single-byte UTF-8 characters are allowed in the folder. A multi-byte character in the folder, can cause parsing to fail, and the message to be rejected.
- Do not use escape strings in the folder. An escape string is treated as the actual value of the element.
- Only Unicode character U+0020 is treated as white space within the folder. All other characters are treated as significant and can cause parsing of the folder to fail, and the message to be rejected.

If parsing of the mq folder fails, or if the folder does not observe these restrictions, the message is rejected with reason code 2527, MQRC_RFH_RESTRICTED_FORMAT_ERR.

Do not add your own properties in the mq folder.

mqema

mqema contains properties that are used only by WebSphere Application Server. The folder has been replaced by mqext.

Do not add your own properties in the mqema folder.

mqext

mqext contains the following types of property:

- Properties that are used only by WebSphere Application Server.
- Properties relating to delayed delivery of messages.

The folder is present if the application has either set at least one of the IBM defined properties or used delivery delay.

Table 524. mqext property name, synonym, data type, and folder			
Property synonym	Property name	Data type	Folder
JMSArmCorrelator	mqext.Arm	string	<mqext><Arm>armCorrelator</Arm></mqext>
JMSRMCorrelator	mqext.Wrm	string	<mqext><Wrm>wrmCorrelator</Wrm></mqext>
JMSDeliveryTime	mqext.Dlt	i8	<mqext><Dlt>DeliveryTime</Dlt></mqext>
JMSDeliveryDelay	mqext.Dly	i8	<mqext><Dly>DeliveryTime</Dly></mqext>

Do not add your own properties in the mqext folder.

mmps

mmps contains properties that are used only by IBM MQ publish/subscribe. The folder is present only if the application has set at least one of the integrated publish/subscribe properties.

Table 525. mmps property name, synonym, data type, and folder			
Property synonym	Property name	Data type	Folder
MQTopicString	mmps.Top	string	<mmps><Top>topicString</Top></mmps>
MQSubUserData	mmps.Sud	string	<mmps><Sud>subscriberUserData</Sud></mmps>
MQIsRetained	mmps.Ret	boolean	<mmps><Ret>isRetained</Ret></mmps>
MQPubOptions	mmps.Pub	i8	<mmps><Pub>publicationOptions</Pub></mmps>
MQPubLevel	mmps.Pbl	i8	<mmps><Pbl>publicationLevel</Pbl></mmps>
MQPubTime	mmpse.Pts	string	<mmps><Pts>publicationTime</Pts></mmps>
MQPubSeqNum	mmpse.Seq	i8	<mmps><Seq>publicationSequenceNumber</Seq></mmps>

Table 525. mqps property name, synonym, data type, and folder (continued)			
Property synonym	Property name	Data type	Folder
MQPubStrIn tData	mqpse.Sid	string	<mqps><Sid>publicationData</Sid></mqps>
MQPubForma t	mqpse.Pfmt	i8	<mqps><Pfmt>messageFormat</Pfmt></mqps>

Do not add your own properties in the mqps folder.

mq_svc

mq_svc contains properties used by SupportPac MA93.

Do not add your own properties in the mq_svc folder.

mqtt

mqtt contains properties use by IBM MQ Telemetry

Table 526. mqtt property name, synonym, data type, and folder			
Property synonym	Property name	Data type	Folder
	mqtt.clientId	string	<mqtt><clientId>topicString</clientId></mqtt>
	mqtt.qos	i4	<mqtt><qos>qualityOfService</qos></mqtt>
	mqtt.msgid	string	<mqtt><msgid>messageIdentifier</msgid></mqtt>

Do not add your own properties in the mqtt folder.

A segmented message put with either MQMF_SEGMENT or MQMF_SEGMENTATION_ALLOWED cannot contain an MQRFH2 with an ungrouped property folder name. The MQPUT fails with reason code 2443, MQRC_SEGMENTATION_NOT_ALLOWED.

Name/value pairs

In the syntax diagram, "Name/value pairs" describes the content of an ordinary folder. An ordinary folder contains groups, and elements. An element is a name/value pair. A group contains elements and other groups.

In terms of trees, elements are leaf nodes, and groups are internal nodes. An internal node, and the folder, which is the root node, can contain a mixture of internal nodes and leaf nodes. A node cannot be both an internal node and a leaf node at the same time; see [Figure 2 on page 508](#).

Properties

In the syntax diagram, "Properties" describes the content of a property folder. A property folder contains groups, and properties. A property is a name/value pair with an optional data type attribute. A group contains properties and other groups.

In terms of trees, properties are leaf nodes, and groups are internal nodes. An internal node, and the property folder, which is the root node, can contain a mixture of internal nodes and leaf nodes. A node cannot be both an internal node and a leaf node at the same time; see [Figure 2 on page 508](#).

Property

A message property is a name/value pair in a property folder. It can optionally include a data type attribute and a property attribute; for an example, see the following code. If the data type attribute is omitted, the property type is string.

```
<pf><p1 dt='i8' > value </p1></pf>
```

The name of a message property is its full path name, with the XML-like, <> syntax, replaced by dots. For example, myPropertyFolder1.myGroup1.myGroup2.myProperty1 is mapped to a *NameValueData* string as follows. The string is formatted for easier reading.

```
<myPropertyFolder1>
  <myGroup1>
    <myGroup2>
      <myProperty1>value</myProperty1>
    </myGroup2>
  </myGroup1>
</myPropertyFolder1>
```

A property folder can contain multiple properties. For example the properties in [Figure 7 on page 515](#) are mapped to the property folder in [Figure 8 on page 515](#)

```
myPropertyFolder1.myProperty4
myPropertyFolder1.myGroup1.myGroup2.myProperty1
myPropertyFolder1.myGroup1.myGroup2.myProperty2
myPropertyFolder1.myGroup1.myProperty3
```

Figure 7. Multiple properties with the same root name

```
<myPropertyFolder1>
  <myProperty4>value</myProperty4>
  <myGroup1>
    <myGroup2>
      <myProperty1>value</myProperty1>
      <myProperty2>value</myProperty2>
    </myGroup2>
    <myProperty3>value</myProperty3>
  </myGroup1>
</myPropertyFolder1>
```

Figure 8. Multiple property name mapping

Name

A name must begin with a *Letter* or an *Underscore*. It must not contain a *Colon*, not end in a *Period* and contain only *Letters*, *Numerals*, *Underscores*, *Hyphens*, and *Dots*. Valid characters are defined in the W3C XML specification, and consist essentially of Unicode categories L1, Lu, Lo, Lt, N1, Mc, Mn, Lm, and Nd ; see [Unicode character categories](#).

The complete path of a property or name/value pair must not break the rule described in “[Invalid path name](#)” on [page 518](#). Paths are restricted to 4095 bytes, must not contain Unicode compatibility characters, and must not start with the string XML.

Group name

A group name has the same syntax as a name. Group names are optional. Properties and name/value pairs can be placed in the root of a folder. Use groups if it helps to organize properties and name/value pairs.

Element name

An element name has the same syntax as a name.

Element value

An element value includes all the white space between the `< Element name >` tag and the `< / Element name >`. Do not use the two characters `<` and `&` in a value. Replace them with `<` and `&`.

Property attributes

The property attributes map property descriptor fields: The mappings are as follows:

Support

<u>sa</u>	MQPD_SUPPORT_OPTIONAL
sr	MQPD_SUPPORT_REQUIRED
sx	MQPD_SUPPORT_REQUIRED_IF_LOCAL

Context

<u>none</u>	MQPD_NO_CONTEXT
user	MQPD_USER _CONTEXT

CopyOptions

forward	MQPD_COPY_FORWARD
reply	MQPD_COPY_REPLY
report	MQPD_COPY_REPORT
publish	MQPD_COPY_PUBLISH
all	MQPD_COPY_ALL
Do not use all in combination with other options.	
default	MQPD_COPY_DEFAULT
Do not use default in combination with other options. default is the same as forward + report + publish	
none	MQPD_COPY_NONE
Do not use none in combination with other options.	

The Support property attributes are only applicable to properties in the mq folder.

The Context and CopyOptions property attributes are applicable to all property folders.

Data types

MQRFH2 data types map to message property types as follows:

Table 527. Data type mappings

MQRFH2 data type	Message property type
bin.hex	MQBYTE[]
boolean	MQBOOL
i1	MQINT8
i2	MQINT16
i4	MQINT32
i8	MQINT64
r4	MQFLOAT32
r8	MQFLOAT64
string	MQCHAR[]

Any element without a data type is assumed to be of type string.

A null value is indicated by the element attribute `xsi:nil='true'`. Do not use the attribute `xsi:nil='false'` for non-null values. For example, the following property has a null value:

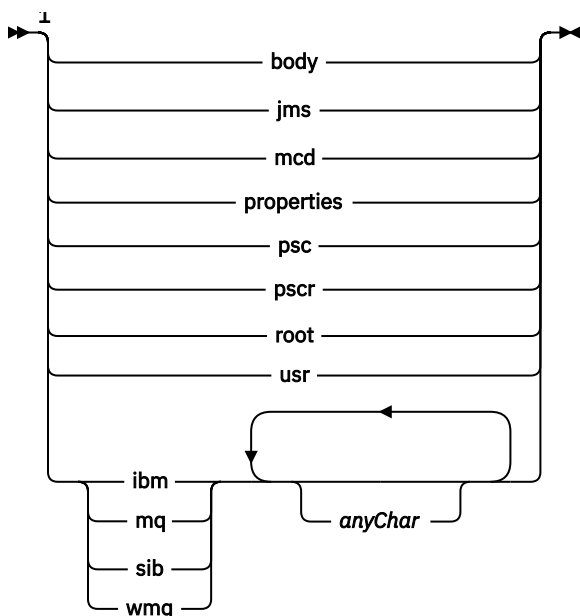
```
<NullProperty
  xsi:nil='true'></NullProperty>
```

A byte or character string property can have an empty value. An empty value is represented by an MQRFH2 element with a zero length element value. For example, the following property has an empty value:

```
<EmptyProperty></EmptyProperty>
```

Reserved folder or property folder name

Restrict the name of a folder or property folder not to start with any of the following strings. The prefixes are reserved for folder or property names created by IBM.

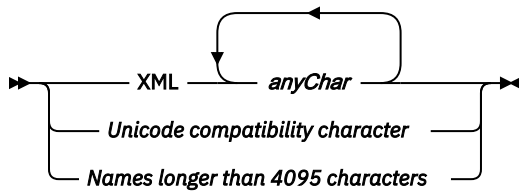


Notes:

- ¹ A reserved folder or property name contains any mixture of lower and uppercase letters.

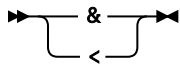
Invalid path name

Restrict the complete path of a name/value pair or a property not to include any of the following strings.



Invalid characters

Always use the escape sequences `&` and `<` instead of the literals `"&"` and `"<"`.

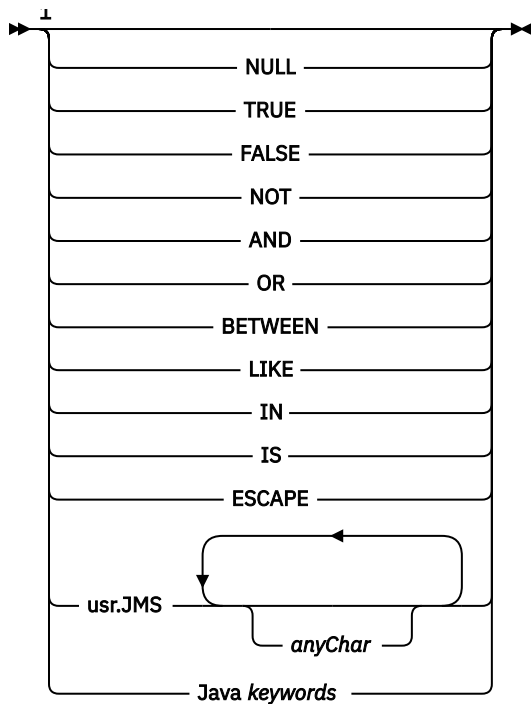


Defined property names

Defined property names are the names of properties that are defined by IBM MQ, or other products, and used by IBM MQ and user applications. Defined properties exist only in defined property folders. Defined property names are described in the description of property folders; see [Property folders](#).

Invalid property name

Do not construct property names that match the following rule. The rule applies to the full property path that names a property, and not only to the property element name.



Notes:

¹ An invalid property name can contain any combination of upper and lowercase.

Invalid attributes

Property folders and properties can include only supported “[Property attributes](#)” on page 516 and “[Data types](#)” on page 516.

Any non-supported XML-like attributes, for example, names with quoted string values, that are included in property folders or properties might be removed.

XML-like attributes included in non-property folders or non-property elements that remain in MQRFH2 headers.

NameValueLength (MQLONG)

The length of the corresponding NameValueData field

This specifies the length in bytes of the data in the *NameValueData* field. *NameValueLength* must be a multiple of four.

Note: The *NameValueLength* and *NameValueData* fields are optional, but if present they must occur as a pair and be adjacent. The pair of fields can be repeated as many times as required, for example:

```
length1 data1 length2 data2 length3 data3
```

Because these fields are optional, they are omitted from the declarations of the structure that are provided for the various programming languages supported.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQRFH_STRUC_ID

Identifier for rules and formatting header structure.

For the C programming language, the constant MQRFH_STRUC_ID_ARRAY is also defined; this has the same value as MQRFH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQRFH_STRUC_ID.

StrucLength (MQLONG)

This is the length in bytes of the MQRFH2 structure, including the *NameValueLength* and *NameValueData* fields at the end of the structure. It is valid for there to be multiple pairs of *NameValueLength* and *NameValueData* fields at the end of the structure, in the sequence:

```
length1, data1, length2, data2, ...
```

StrucLength does *not* include any user data that might follow the last *NameValueData* field at the end of the structure.

To avoid problems with converting the user data in some environments, *StrucLength* must be a multiple of four.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *NameValueLength* and *NameValueData* fields:

MQRFH_STRUC_LENGTH_FIXED_2

Length of fixed part of MQRFH2 structure.

The initial value of this field is MQRFH_STRUC_LENGTH_FIXED_2.

Version (MQLONG)

This is the structure version number; the value must be:

MQRFH_VERSION_2

Version-2 rules and formatting header structure.

The initial value of this field is MQRFH_VERSION_2.

Initial values and language declarations for MQRFH2

Table 528. Initial values of fields in MQRFH2 for MQRFH2		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQRFH_STRUC_ID	'RFH¬'
<i>Version</i>	MQRFH_VERSION_2	2
<i>StrucLength</i>	MQRFH_STRUC_LENGTH_FIXED_2	36
<i>Encoding</i>	MQENC_NATIVE	Depends on environment
<i>CodedCharSetId</i>	MQCCSI_INHERIT	-2
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQRFH_NONE	0
<i>NameValueCCSID</i>	None	1208
Notes: <ol style="list-style-type: none"> 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQRFH2_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre> MQRFH2 MyRFH2 = {MQRFH2_DEFAULT}; </pre> 		

C declaration

```
typedef struct tagMQRFH2 MQRFH2;
struct tagMQRFH2 {
    MQCHAR4  StrucId;          /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   StrucLength;      /* Total length of MQRFH2 including all
                               NameValueLength and NameValueData
                               fields */
    MQLONG   Encoding;         /* Numeric encoding of data that follows
                               last NameValueData field */
    MQLONG   CodedCharSetId;   /* Character set identifier of data that
                               follows last NameValueData field */
    MQCHAR8  Format;           /* Format name of data that follows last
                               NameValueData field */
    MQLONG   Flags;            /* Flags */
    MQLONG   NameValueCCSID;   /* Character set identifier of
                               NameValueData */
};
```

COBOL declaration

```
**  MQRFH2 structure
10 MQRFH2.
**  Structure identifier
15 MQRFH2-STRUCID      PIC X(4).
**  Structure version number
15 MQRFH2-VERSION      PIC S9(9) BINARY.
**  Total length of MQRFH2 including all NAMEVALUELENGTH and
**  NAMEVALUEDATA fields
15 MQRFH2-STRUCLNGTH   PIC S9(9) BINARY.
**  Numeric encoding of data that follows last NAMEVALUEDATA field
15 MQRFH2-ENCODING     PIC S9(9) BINARY.
**  Character set identifier of data that follows last NAMEVALUEDATA
**  field
15 MQRFH2-CODEDCHARSETID PIC S9(9) BINARY.
**  Format name of data that follows last NAMEVALUEDATA field
15 MQRFH2-FORMAT       PIC X(8).
**  Flags
15 MQRFH2-FLAGS        PIC S9(9) BINARY.
```

```

**      Character set identifier of NAMEVALUEDATA
15 MQRFH2-NAMEVALUECCSID PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQRFH2 based,
3 StrucId      char(4),      /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 StrucLength   fixed bin(31), /* Total length of MQRFH2 including
                                all NameValueLength and
                                NameValueData fields */
3 Encoding      fixed bin(31), /* Numeric encoding of data that
                                follows last NameValueData field */
3 CodedCharSetId fixed bin(31), /* Character set identifier of data
                                that follows last NameValueData
                                field */
3 Format         char(8),      /* Format name of data that follows
                                last NameValueData field */
3 Flags         fixed bin(31), /* Flags */
3 NameValueCCSID fixed bin(31); /* Character set identifier of
                                NameValueData */

```

High Level Assembler declaration

```

MQRFH          DSECT
MQRFH_STRUCID  DS   CL4  Structure identifier
MQRFH_VERSION  DS   F    Structure version number
MQRFH_STRUCLNGTH DS   F    Total length of MQRFH2 including all
*                                NAMEVALUELENGTH and NAMEVALUEDATA fields
MQRFH_ENCODING DS   F    Numeric encoding of data that follows
*                                last NAMEVALUEDATA field
MQRFH_CODEDCHARSETID DS   F    Character set identifier of data that
*                                follows last NAMEVALUEDATA field
MQRFH_FORMAT   DS   CL8  Format name of data that follows last
*                                NAMEVALUEDATA field
MQRFH_FLAGS    DS   F    Flags
MQRFH_NAMEVALUECCSID DS   F    Character set identifier of
*                                NAMEVALUEDATA
*
MQRFH_LENGTH   EQU   *-MQRFH
                ORG   MQRFH
MQRFH_AREA     DS    CL(MQRFH_LENGTH)

```

Visual Basic declaration

```

Type MQRFH2
StrucId      As String*4 'Structure identifier'
Version      As Long     'Structure version number'
StrucLength   As Long     'Total length of MQRFH2 including all'
                                'NameValueLength and NameValueData fields'
Encoding      As Long     'Numeric encoding of data that follows'
                                'last NameValueData field'
CodedCharSetId As Long     'Character set identifier of data that'
                                'follows last NameValueData field'
Format        As String*8 'Format name of data that follows last'
                                'NameValueData field'
Flags         As Long     'Flags'
NameValueCCSID As Long     'Character set identifier of NameValueData'
End Type

```

MQRMH - Reference message header

The following table summarizes the fields in the structure.

Table 529. Fields in MQRMH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version

Table 529. Fields in MQRMH (continued)

Field	Description	Topic
<i>StrucLength</i>	Total length of MQRMH, including strings at end of fixed fields, but not the bulk data	StrucLength
<i>Encoding</i>	Numeric encoding of bulk data	Encoding
<i>CodedCharSetId</i>	Character set identifier of bulk data	CodedCharSetId
<i>Format</i>	Format name of bulk data	Format
<i>Flags</i>	Reference message flags	Flags
<i>ObjectType</i>	Object type	ObjectType
<i>ObjectInstanceId</i>	Object instance identifier	ObjectInstanceId
<i>SrcEnvLength</i>	Length of source environment data	SrcEnvLength
<i>SrcEnvOffset</i>	Offset of source environment data	SrcEnvOffset
<i>SrcNameLength</i>	Length of source object name	SrcNameLength
<i>SrcNameOffset</i>	Offset of source object name	SrcNameOffset
<i>DestEnvLength</i>	Length of destination environment data	DestEnvLength
<i>DestEnvOffset</i>	Offset of destination environment data	DestEnvOffset
<i>DestNameLength</i>	Length of destination object name	DestNameLength
<i>DestNameOffset</i>	Offset of destination object name	DestNameOffset
<i>DataLogicalLength</i>	Length of bulk data	DataLogicalLength
<i>DataLogicalOffset</i>	Low offset of bulk data	DataLogicalOffset
<i>DataLogicalOffset2</i>	High offset of bulk data	DataLogicalOffset2

Overview for MQRMH

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

Purpose: The MQRMH structure defines the format of a reference message header. This header is used with user-written message channel exits to send extremely large amounts of data (called *bulk data*) from one queue manager to another. The difference compared to normal messaging is that the bulk data is not stored on a queue; instead, only a *reference* to the bulk data is stored on the queue. This reduces the possibility of MQ resources being exhausted by a small number of extremely large messages.

Format name: MQFMT_REF_MSG_HEADER.

Character set and encoding: Character data in MQRMH, and the strings addressed by the offset fields, must be in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in MQRMH must be in the native machine encoding; this is given by the value of MQENC_NATIVE for the C programming language.

Set the character set and encoding of the MQRMH into the *CodedCharSetId* and *Encoding* fields in:

- The MQMD (if the MQRMH structure is at the start of the message data), or
- The header structure that precedes the MQRMH structure (all other cases).

Usage: An application puts a message consisting of an MQRMH, but omitting the bulk data. When a message channel agent (MCA) reads the message from the transmission queue, a user-supplied message exit is invoked to process the reference message header. The exit can append to the reference message

the bulk data identified by the MQRMH structure, before the MCA sends the message through the channel to the next queue manager.

At the receiving end, a message exit that waits for reference messages must exist. When a reference message is received, the exit must create the object from the bulk data that follows the MQRMH in the message, and then pass on the reference message without the bulk data. The reference message can later be retrieved by an application reading the reference message (without the bulk data) from a queue.

Normally, the MQRMH structure is all that is in the message. However, if the message is on a transmission queue, one or more additional headers precede the MQRMH structure.

A reference message can also be sent to a distribution list. In this case, the MQDH structure and its related records precede the MQRMH structure when the message is on a transmission queue.

Note: Do not send a reference message as a segmented message, because the message exit cannot process it correctly.

Data conversion: For data conversion purposes, converting the MQRMH structure includes conversion of the source environment data, source object name, destination environment data, and destination object name. Any other bytes within *StrucLength* bytes of the start of the structure are either discarded or have undefined values after data conversion. The bulk data is converted provided that all the following are true:

- The bulk data is present in the message when the data conversion is performed.
- The *Format* field in MQRMH has a value other than MQFMT_NONE.
- A user-written data-conversion exit exists with the format name specified.

Be aware, however, that usually the bulk data is *not* present in the message when the message is on a queue, and that as a result the bulk data is converted by the MQGMO_CONVERT option.

Fields for MQRMH

The MQRMH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This specifies the character set identifier of the bulk data; it does not apply to character data in the MQRMH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

MQCCSI_INHERIT

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

Do not use MQCCSI_INHERIT if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

This value is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

The initial value of this field is MQCCSI_UNDEFINED.

DataLogicalLength (MQLONG)

The *DataLogicalLength* field specifies the length of the bulk data referenced by the MQRMH structure.

If the bulk data is actually present in the message, the data begins at an offset of *StrucLength* bytes from the start of the MQRMH structure. The length of the entire message minus *StrucLength* gives the length of the bulk data present.

If data is present in the message, *DataLogicalLength* specifies the amount of that data that is relevant. The normal case is for *DataLogicalLength* to have the same value as the length of data present in the message.

If the MQRMH structure represents the remaining data in the object (starting from the specified logical offset), you can use the value zero for *DataLogicalLength*, provided that the bulk data is not actually present in the message.

If no data is present, the end of MQRMH coincides with the end of the message.

The initial value of this field is 0.

DataLogicalOffset (MQLONG)

This field specifies the low offset of the bulk data from the start of the object of which the bulk data forms part. The offset of the bulk data from the start of the object is called the *logical offset*. This is *not* the physical offset of the bulk data from the start of the MQRMH structure; that offset is given by *StrucLength*.

To allow large objects to be sent using reference messages, the logical offset is divided into two fields, and the actual logical offset is given by the sum of these two fields:

- *DataLogicalOffset* represents the remainder obtained when the logical offset is divided by 1 000 000 000. It is thus a value in the range 0 through 999 999 999.
- *DataLogicalOffset2* represents the result obtained when the logical offset is divided by 1 000 000 000. It is thus the number of complete multiples of 1 000 000 000 that exist in the logical offset. The number of multiples is in the range 0 through 999 999 999.

The initial value of this field is 0.

DataLogicalOffset2 (MQLONG)

This field specifies the high offset of the bulk data from the start of the object of which the bulk data forms part. It is a value in the range 0 through 999 999 999. See *DataLogicalOffset* for details.

The initial value of this field is 0.

DestEnvLength (MQLONG)

This is the length of the destination environment data. If this field is zero, there is no destination environment data, and *DestEnvOffset* is ignored.

DestEnvOffset (MQLONG)

This field specifies the offset of the destination environment data from the start of the MQRMH structure. Destination environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, on Windows the destination environment data might be the directory path of the object where the bulk data is to be stored. However, if the creator does not know the destination environment data, it is the responsibility of the user-supplied message exit to determine any environment information needed.

The length of the destination environment data is given by *DestEnvLength* ; if this length is zero, there is no destination environment data, and *DestEnvOffset* is ignored. If present, the destination environment data must reside completely within *StrucLength* bytes from the start of the structure.

Applications must not assume that the destination environment data is contiguous with any of the data addressed by the *SrcEnvOffset*, *SrcNameOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

DestNameLength (MQLONG)

The length of the destination object name. If this field is zero, there is no destination object name, and *DestNameOffset* is ignored.

DestNameOffset (MQLONG)

This field specifies the offset of the destination object name from the start of the MQRMH structure. The destination object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the destination object name, it is the responsibility of the user-supplied message exit to identify the object to be created or modified.

The length of the destination object name is given by *DestNameLength* ; if this length is zero, there is no destination object name, and *DestNameOffset* is ignored. If present, the destination object name must reside completely within *StrucLength* bytes from the start of the structure.

Applications must not assume that the destination object name is contiguous with any of the data addressed by the *SrcEnvOffset*, *SrcNameOffset*, and *DestEnvOffset* fields.

The initial value of this field is 0.

Encoding (MQLONG)

This specifies the numeric encoding of the bulk data; it does not apply to numeric data in the MQRMH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is MQENC_NATIVE.

Flags (MQLONG)

These are reference message flags. The following flags are defined:

MQRMHF_LAST

This flag indicates that the reference message represents or contains the last part of the referenced object.

MQRMHF_NOT_LAST

Reference message does not contain or represent last part of object. MQRMHF_NOT_LAST aids program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQRMHF_NOT_LAST.

Format (MQCHAR8)

This specifies the format name of the bulk data.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *Format* field in MQMD.

The initial value of this field is MQFMT_NONE.

ObjectInstanceId (MQBYTE24)

Use this field to identify a specific instance of an object. If it is not needed, set it to the following value:

MQOII_NONE

No object instance identifier specified. The value is binary zero for the length of the field.

For the C programming language, the constant MQOII_NONE_ARRAY is also defined; this has the same value as MQOII_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_OBJECT_INSTANCE_ID_LENGTH. The initial value of this field is MQOII_NONE.

ObjectType (MQCHAR8)

This is a name that the message exit can use to recognize types of reference message that it supports. The name must conform to the same rules as the *Format* field, see [“Format \(MQCHAR8\)” on page 525](#).

The initial value of this field is 8 blanks.

SrcEnvLength (MQLONG)

The length of the source environment data. If this field is zero, there is no source environment data, and *SrcEnvOffset* is ignored.

The initial value of this field is 0.

SrcEnvOffset (MQLONG)

This field specifies the offset of the source environment data from the start of the MQRMH structure. Source environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, on Windows the source environment data might be the directory path of the object containing the bulk data. However, if the creator does not know the source environment data, the user-supplied message exit must determine any environment information needed.

The length of the source environment data is given by *SrcEnvLength* ; if this length is zero, there is no source environment data, and *SrcEnvOffset* is ignored. If present, the source environment data must reside completely within *StrucLength* bytes from the start of the structure.

Applications must not assume that the environment data starts immediately after the last fixed field in the structure or that it is contiguous with any of the data addressed by the *SrcNameOffset*, *DestEnvOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

SrcNameLength (MQLONG)

The length of the source object name. If this field is zero, there is no source object name, and *SrcNameOffset* is ignored.

The initial value of this field is 0.

SrcNameOffset (MQLONG)

This field specifies the offset of the source object name from the start of the MQRMH structure. The source object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the source object name, the user-supplied message exit must identify the object to be accessed.

The length of the source object name is given by *SrcNameLength* ; if this length is zero, there is no source object name, and *SrcNameOffset* is ignored. If present, the source object name must reside completely within *StrucLength* bytes from the start of the structure.

Applications must not assume that the source object name is contiguous with any of the data addressed by the *SrcEnvOffset*, *DestEnvOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQRMH_STRUC_ID

Identifier for reference message header structure.

For the C programming language, the constant MQRMH_STRUC_ID_ARRAY is also defined; this has the same value as MQRMH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQRMH_STRUC_ID.

StrucLength (MQLONG)

The total length of MQRMH, including strings at the end of fixed fields, but not the bulk data.

The initial value of this field is zero.

Version (MQLONG)

The structure version number. The value must be:

MQRMH_VERSION_1

Version-1 reference message header structure.

The following constant specifies the version number of the current version:

MQRMH_CURRENT_VERSION

Current version of reference message header structure.

The initial value of this field is MQRMH_VERSION_1.

Initial values and language declarations for MQRMH

Table 530. Initial values of fields in MQRMH for MQRMH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQRMH_STRUC_ID	'RMH↵'
<i>Version</i>	MQRMH_VERSION_1	1
<i>StrucLength</i>	None	0
<i>Encoding</i>	MQENC_NATIVE	Depends on environment
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQRMHF_NOT_LAST	0
<i>ObjectType</i>	None	Blanks
<i>ObjectInstanceId</i>	MQOII_NONE	Nulls
<i>SrcEnvLength</i>	None	0
<i>SrcEnvOffset</i>	None	0
<i>SrcNameLength</i>	None	0
<i>SrcNameOffset</i>	None	0
<i>DestEnvLength</i>	None	0
<i>DestEnvOffset</i>	None	0
<i>DestNameLength</i>	None	0
<i>DestNameOffset</i>	None	0
<i>DataLogicalLength</i>	None	0
<i>DataLogicalOffset</i>	None	0
<i>DataLogicalOffset2</i>	None	0
Notes:		
1. The symbol ↵ represents a single blank character.		
2. In the C programming language, the macro variable MQRMH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:		
<pre>MQRMH MyRMH = {MQRMH_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQRMH MQRMH;
struct tagMQRMH {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;          /* Structure version number */
    MQLONG    StrucLength;       /* Total length of MQRMH, including
                                strings at end of fixed fields, but
                                not the bulk data */
    MQLONG    Encoding;         /* Numeric encoding of bulk data */
    MQLONG    CodedCharSetId;   /* Character set identifier of bulk
                                data */
    MQCHAR8   Format;           /* Format name of bulk data */
    MQLONG    Flags;            /* Reference message flags */
    MQCHAR8   ObjectType;       /* Object type */
    MQBYTE24  ObjectInstanceId; /* Object instance identifier */
    MQLONG    SrcEnvLength;     /* Length of source environment data */
    MQLONG    SrcEnvOffset;     /* Offset of source environment data */
    MQLONG    SrcNameLength;    /* Length of source object name */
    MQLONG    SrcNameOffset;    /* Offset of source object name */
    MQLONG    DestEnvLength;    /* Length of destination environment
                                data */
    MQLONG    DestEnvOffset;    /* Offset of destination environment
                                data */
    MQLONG    DestNameLength;   /* Length of destination object name */
    MQLONG    DestNameOffset;   /* Offset of destination object name */
    MQLONG    DataLogicalLength; /* Length of bulk data */
    MQLONG    DataLogicalOffset; /* Low offset of bulk data */
    MQLONG    DataLogicalOffset2; /* High offset of bulk data */
};
```

COBOL declaration

```
**      MQRMH structure
10 MQRMH.
**      Structure identifier
15 MQRMH-STRUCID          PIC X(4).
**      Structure version number
15 MQRMH-VERSION          PIC S9(9) BINARY.
**      Total length of MQRMH, including strings at end of fixed fields,
**      but not the bulk data
15 MQRMH-STRUCLength     PIC S9(9) BINARY.
**      Numeric encoding of bulk data
15 MQRMH-ENCODING        PIC S9(9) BINARY.
**      Character set identifier of bulk data
15 MQRMH-CODEDCHARSETID  PIC S9(9) BINARY.
**      Format name of bulk data
15 MQRMH-FORMAT          PIC X(8).
**      Reference message flags
15 MQRMH-FLAGS           PIC S9(9) BINARY.
**      Object type
15 MQRMH-OBJECTTYPE      PIC X(8).
**      Object instance identifier
15 MQRMH-OBJECTINSTANCEID PIC X(24).
**      Length of source environment data
15 MQRMH-SRCENVLENGTH    PIC S9(9) BINARY.
**      Offset of source environment data
15 MQRMH-SRCENVOFFSET    PIC S9(9) BINARY.
**      Length of source object name
15 MQRMH-SRCNAMELENGTH   PIC S9(9) BINARY.
**      Offset of source object name
15 MQRMH-SRCNAMEOFFSET   PIC S9(9) BINARY.
**      Length of destination environment data
15 MQRMH-DESTENVLENGTH   PIC S9(9) BINARY.
**      Offset of destination environment data
15 MQRMH-DESTENVOFFSET   PIC S9(9) BINARY.
**      Length of destination object name
15 MQRMH-DESTNAMELENGTH  PIC S9(9) BINARY.
**      Offset of destination object name
15 MQRMH-DESTNAMEOFFSET  PIC S9(9) BINARY.
**      Length of bulk data
15 MQRMH-DATALOGICALLength PIC S9(9) BINARY.
**      Low offset of bulk data
15 MQRMH-DATALOGICALOFFSET PIC S9(9) BINARY.
**      High offset of bulk data
15 MQRMH-DATALOGICALOFFSET2 PIC S9(9) BINARY.
```

PL/I declaration

```

dcl
  1 MQRMH based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31), /* Structure version number */
    3 StrucLength      fixed bin(31), /* Total length of MQRMH,
                                     including strings at end of
                                     fixed fields, but not the bulk
                                     data */
    3 Encoding         fixed bin(31), /* Numeric encoding of bulk
                                     data */
    3 CodedCharSetId   fixed bin(31), /* Character set identifier of
                                     bulk data */
    3 Format            char(8),          /* Format name of bulk data */
    3 Flags            fixed bin(31), /* Reference message flags */
    3 ObjectType       char(8),          /* Object type */
    3 ObjectInstanceId char(24),         /* Object instance identifier */
    3 SrcEnvLength     fixed bin(31), /* Length of source environment
                                     data */
    3 SrcEnvOffset     fixed bin(31), /* Offset of source environment
                                     data */
    3 SrcNameLength    fixed bin(31), /* Length of source object name */
    3 SrcNameOffset    fixed bin(31), /* Offset of source object name */
    3 DestEnvLength    fixed bin(31), /* Length of destination
                                     environment data */
    3 DestEnvOffset    fixed bin(31), /* Offset of destination
                                     environment data */
    3 DestNameLength   fixed bin(31), /* Length of destination object
                                     name */
    3 DestNameOffset   fixed bin(31), /* Offset of destination object
                                     name */
    3 DataLogicalLength fixed bin(31), /* Length of bulk data */
    3 DataLogicalOffset fixed bin(31), /* Low offset of bulk data */
    3 DataLogicalOffset2 fixed bin(31); /* High offset of bulk data */

```

High Level Assembler declaration

MQRMH	DSECT		
MQRMH_STRUCID	DS	CL4	Structure identifier
MQRMH_VERSION	DS	F	Structure version number
MQRMH_STRUCLNGTH	DS	F	Total length of MQRMH, including
*			strings at end of fixed fields, but
*			not the bulk data
MQRMH_ENCODING	DS	F	Numeric encoding of bulk data
MQRMH_CODEDCHARSETID	DS	F	Character set identifier of bulk
*			data
MQRMH_FORMAT	DS	CL8	Format name of bulk data
MQRMH_FLAGS	DS	F	Reference message flags
MQRMH_OBJECTTYPE	DS	CL8	Object type
MQRMH_OBJECTINSTANCEID	DS	XL24	Object instance identifier
MQRMH_SRCENVLENGTH	DS	F	Length of source environment data
MQRMH_SRCENVOFFSET	DS	F	Offset of source environment data
MQRMH_SRCNAMELENGTH	DS	F	Length of source object name
MQRMH_SRCNAMEOFFSET	DS	F	Offset of source object name
MQRMH_DESTENVLENGTH	DS	F	Length of destination environment
*			data
MQRMH_DESTENVOFFSET	DS	F	Offset of destination environment
*			data
MQRMH_DESTNAMELENGTH	DS	F	Length of destination object name
MQRMH_DESTNAMEOFFSET	DS	F	Offset of destination object name
MQRMH_DATALOGICALENGTH	DS	F	Length of bulk data
MQRMH_DATALOGICALOFFSET	DS	F	Low offset of bulk data
MQRMH_DATALOGICALOFFSET2	DS	F	High offset of bulk data
*			
MQRMH_LENGTH	EQU	*-MQRMH	
	ORG	MQRMH	
MQRMH_AREA	DS	CL(MQRMH_LENGTH)	

Visual Basic declaration

Type MQRMH		
StrucId	As String*4	'Structure identifier'
Version	As Long	'Structure version number'
StrucLength	As Long	'Total length of MQRMH, including'
		'strings at end of fixed fields, but'
		'not the bulk data'

```

Encoding           As Long      'Numeric encoding of bulk data'
CodedCharSetId     As Long      'Character set identifier of bulk data'
Format             As String*8  'Format name of bulk data'
Flags              As Long      'Reference message flags'
ObjectType         As String*8  'Object type'
ObjectInstanceId   As MQBYTE24  'Object instance identifier'
SrcEnvLength       As Long      'Length of source environment data'
SrcEnvOffset       As Long      'Offset of source environment data'
SrcNameLength      As Long      'Length of source object name'
SrcNameOffset      As Long      'Offset of source object name'
DestEnvLength      As Long      'Length of destination environment
                                'data'
DestEnvOffset      As Long      'Offset of destination environment
                                'data'
DestNameLength     As Long      'Length of destination object name'
DestNameOffset     As Long      'Offset of destination object name'
DataLogicalLength  As Long      'Length of bulk data'
DataLogicalOffset  As Long      'Low offset of bulk data'
DataLogicalOffset2 As Long      'High offset of bulk data'
End Type

```

MQRR - Response record

The following table summarizes the fields in the structure.

Table 531. Fields in MQRR		
Field	Description	Topic
<i>CompCode</i>	Completion code for queue	CompCode
<i>Reason</i>	Reason code for queue	Reason

Overview for MQRR

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

Purpose: Use the MQRR structure to receive the completion code and reason code resulting from the open or put operation for a single destination queue, when the destination is a distribution list. MQRR is an output structure for the MQOPEN, MQPUT, and MQPUT1 calls.

Character set and encoding: Data in MQRR must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQOPEN and MQPUT calls, or on the MQPUT1 call, you can determine the completion codes and reason codes for all the queues in a distribution list when the outcome of the call is mixed, that is, when the call succeeds for some queues in the list but fails for others. Reason code MQRC_MULTIPLE_REASONS from the call indicates that the response records (if provided by the application) have been set by the queue manager.

Fields for MQRR

The MQRR structure contains the following fields; the fields are described in **alphabetical order**:

CompCode (MQLONG)

This is the completion code resulting from the open or put operation for the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is MQCC_OK.

Reason (MQLONG)

This is the reason code resulting from the open or put operation for the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is MQRC_NONE.

Initial values and language declarations for MQRR

Table 532. Initial values of fields in MQRR for MQRR		
Field name	Name of constant	Value of constant
CompCode	MQCC_OK	0
Reason	MQRC_NONE	0
Notes: 1. In the C programming language, the macro variable MQRR_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQRR MyRR = {MQRR_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQRR MQRR;  
struct tagMQRR {  
    MQLONG  CompCode; /* Completion code for queue */  
    MQLONG  Reason;   /* Reason code for queue */  
};
```

COBOL declaration

```
**  MQRR structure  
10  MQRR.  
**    Completion code for queue  
15  MQRR-COMPCODE PIC S9(9) BINARY.  
**    Reason code for queue  
15  MQRR-REASON  PIC S9(9) BINARY.
```

PL/I declaration

```
dcl  
  1 MQRR based,  
    3 CompCode fixed bin(31), /* Completion code for queue */  
    3 Reason   fixed bin(31); /* Reason code for queue */
```

Visual Basic declaration

```
Type MQRR  
    CompCode As Long 'Completion code for queue'  
    Reason   As Long 'Reason code for queue'  
End Type
```

MQSCO - SSL configuration options

The following table summarizes the fields in the structure.

Table 533. Fields in MQSCO		
Field	Description	Topic
StrucId	Structure identifier	StrucId

Table 533. Fields in MQSCO (continued)		
Field	Description	Topic
<i>Version</i>	Structure version number	Version
<i>KeyRepository</i>	Location of key repository	KeyRepository
<i>CryptoHardware</i>	Details of cryptographic hardware	CryptoHardware
<i>AuthInfoRecCount</i>	Number of MQAIR records present	AuthInfoRecCount
<i>AuthInfoRecOffset</i>	Offset of first MQAIR record from start of MQSCO	AuthInfoRecOffset
<i>AuthInfoRecPtr</i>	Address of first MQAIR record	AuthInfoRecPtr
Note: The following two fields are ignored if <i>Version</i> is less than MQSCO_VERSION_2.		
<i>KeyResetCount</i>	SSL secret key reset count	KeyResetCount
<i>FipsRequired</i>	Use FIPS-certified cryptographic algorithms in IBM MQ	“FipsRequired (MQLONG)” on page 534
Note: The following field is ignored if <i>Version</i> is less than MQSCO_VERSION_3.		
<i>EncryptionPolicySuiteB</i>	Use only Suite B cryptographic algorithms	EncryptionPolicySuiteB
Note: The following field is ignored if <i>Version</i> is less than MQSCO_VERSION_4.		
<i>CertificateValPolicy</i>	Certificate validation policy	CertificateValPolicy
Note: The following field is ignored if <i>Version</i> is less than MQSCO_VERSION_5.		
<i>CertificateLabel</i>	Details the certificate label that is being used.	CertificateLabel

Related reference

[“MQCNO - Connect options” on page 306](#)

The following table summarizes the fields in the structure.

[“Overview for MQSCO” on page 532](#)

Availability: AIX, HP-UX, IBM i, Solaris, Linux and Windows clients.

[“Fields for MQSCO” on page 532](#)

[“Initial values and language declarations for MQSCO” on page 536](#)

Overview for MQSCO

Availability: AIX, HP-UX, IBM i, Solaris, Linux and Windows clients.

Purpose: The MQSCO structure (in conjunction with the SSL fields in the MQCD structure) allows an application running as an IBM MQ MQI client to specify configuration options that control the use of SSL for the client connection when the channel protocol is TCP/IP. The structure is an input parameter on the MQCONN call.

If the channel protocol for the client channel is not TCP/IP, the MQSCO structure is ignored.

Character set and encoding: Data in MQSCO must be in the character set given by the *CodedCharSetId* queue-manager attribute, and encoding of the local queue manager given by MQENC_NATIVE.

Fields for MQSCO

The MQSCO structure contains the following fields; the fields are described in **alphabetical order**:

AuthInfoRecCount (MQLONG)

This is the number of authentication information (MQAIR) records addressed by the *AuthInfoRecPtr* or *AuthInfoRecOffset* fields. For more information, see [“MQAIR - Authentication information record” on page 263](#). The value must be zero or greater. If the value is not valid, the call fails with reason code MQRC_AUTH_INFO_REC_COUNT_ERROR.

This is an input field. The initial value of this field is 0.

AuthInfoRecOffset (MQLONG)

This is the offset in bytes of the first authentication information record from the start of the MQSCO structure. The offset can be positive or negative. The field is ignored if *AuthInfoRecCount* is zero.

You can use either *AuthInfoRecOffset* or *AuthInfoRecPtr* to specify the MQAIR records, but not both; see the description of the *AuthInfoRecPtr* field for details.

This is an input field. The initial value of this field is 0.

AuthInfoRecPtr (PMQAIR)

This is the address of the first authentication information record. The field is ignored if *AuthInfoRecCount* is zero.

You can provide the array of MQAIR records in one of two ways:

- By using the pointer field *AuthInfoRecPtr*

In this case, the application can declare an array of MQAIR records that is separate from the MQSCO structure, and set *AuthInfoRecPtr* to the address of the array.

Consider using *AuthInfoRecPtr* for programming languages that support the pointer data type in a fashion that is portable to different environments (for example, the C programming language).

- By using the offset field *AuthInfoRecOffset*

In this case, the application must declare a compound structure containing an MQSCO followed by the array of MQAIR records, and set *AuthInfoRecOffset* to the offset of the first record in the array from the start of the MQSCO structure. Ensure that this value is correct, and has a value that can be accommodated within an MQLONG (the most restrictive programming language is COBOL, for which the valid range is -999 999 999 through +999 999 999).

Consider using *AuthInfoRecOffset* for programming languages that do not support the pointer data type, or that implement the pointer data type in a fashion that is not portable to different environments (for example, the COBOL programming language).

Whatever technique you choose, only one of *AuthInfoRecPtr* and *AuthInfoRecOffset* can be used; the call fails with reason code MQRC_AUTH_INFO_REC_ERROR if both are nonzero.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

CertificateLabel (MQCHAR64)

This field gives details of the certificate label being used.

IBM MQ initializes the default value for the *CertificateLabel* field as blanks.

This is interpreted at runtime as the default value, and is backwards compatible.

For example, specifying a MQSCO version less than 5.0, or using the default value of blanks for the *CertificateLabel* field, uses the preexisting default value of `ibmwebspheremq<user_id>`.

CertificateValPolicy (MQLONG)

This field specifies what type of certificate validation policy is used. The field can be set to one of the following values:

MQ_CERT_VAL_POLICY_ANY

Apply each of the certificate validation policies supported by the secure sockets library. Accept the certificate chain if any of the policies considers the certificate chain valid.

MQ_CERT_VAL_POLICY_RFC5280

Apply only the RFC5280 compliant certificate validation policy. This setting provides stricter validation than the ANY setting, but rejects some older digital certificates.

The initial value of this field is MQ_CERT_VAL_POLICY_ANY

CryptoHardware (MQCHAR256)

This field gives configuration details for cryptographic hardware connected to the client system.

Set the field to a string of the following format, or leave it blank or null:

```
GSK_PKCS11=< the PKCS #11 driver path and file name >;< the PKCS #11  
token label >;< the PKCS #11 token password >; <symmetric cipher setting>;
```

To use cryptographic hardware which conforms to the PKCS #11 interface, for example, the IBM 4960 or IBM 4764, the PKCS #11 driver path, PKCS #11 token label, and PKCS #11 token password strings must be specified, each terminated by a semi-colon.

The PKCS #11 driver path is an absolute path to the shared library providing support for the PKCS #11 card. The PKCS #11 driver file name is the name of the shared library. An example of the value required for the PKCS #11 path and file name is:

```
/usr/lib/pkcs11/PKCS11_API.so
```

The PKCS #11 token label must match the label you configured your hardware with.

If no cryptographic hardware configuration is required, set the field to blank or null.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. If the value is not valid, or leads to a failure when used to configure the cryptographic hardware, the call fails with reason code MQRC_CRYPTOHARDWARE_ERROR.

This is an input field. The length of this field is given by MQ_SSL_CRYPTOHARDWARE_LENGTH. The initial value of this field is the null string in C, and blank characters in other programming languages.

EncryptionPolicySuiteB(MQLONG)

This field Specifies whether Suite B compliant cryptography is used and what level of strength is employed. The value can be one or more of:

- MQ_SUITE_B_NONE
Suite B compliant cryptography is not used.
- MQ_SUITE_B_128_BIT
Suite B 128-bit strength security is used.
- MQ_SUITE_B_192_BIT
Suite B 192-bit strength security is used.

Note: Using the MQ_SUITE_B_NONE with any other value in this field is invalid.

FipsRequired (MQLONG)

IBM MQ can be configured with cryptographic hardware so that the cryptography modules used are those provided by the hardware product; these can be FIPS-certified to a particular level depending on the cryptographic hardware product in use. Use this field to specify that only FIPS-certified algorithms are used if the cryptography is provided in IBM MQ-provided software.

Note: On UNIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the [IBM Crypto for C \(ICC\) certificate](#) and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the [NIST CMVP modules in process list](#).

When IBM MQ is installed an implementation of SSL cryptography is also installed which provides some FIPS-certified modules.

The values can be:

MQSSL_FIPS_NO

This is the default value. When set to this value:

- Any CipherSpec supported on a particular platform can be used.
- If run without use of cryptographic hardware, the following CipherSpecs run using FIPS 140-2 certified cryptography on the IBM MQ platforms:
 - TLS_RSA_WITH_3DES_EDE_CBC_SHA
 - TLS_RSA_WITH_AES_128_CBC_SHA
 - TLS_RSA_WITH_AES_256_CBC_SHA

MQSSL_FIPS_YES

When set to this value, unless you are using cryptographic hardware to perform the cryptography, you can be sure that

- Only FIPS-certified cryptographic algorithms can be used in the CipherSpec applying to this client connection.
- Inbound and outbound SSL channel connections only succeed if one of the following Cipher Specs are used:
 - TLS_RSA_WITH_3DES_EDE_CBC_SHA
 - TLS_RSA_WITH_AES_128_CBC_SHA
 - TLS_RSA_WITH_AES_256_CBC_SHA

Notes:

1. CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated.
2. Where possible, if FIPS-only CipherSpecs are configured then the MQI client rejects connections which specify a non-FIPS CipherSpec with MQRC_SSL_INITIALIZATION_ERROR. IBM MQ does not guarantee to reject all such connections and it is your responsibility to determine whether your IBM MQ configuration is FIPS-compliant.

KeyRepository (MQCHAR256)

This field is relevant only for IBM MQ MQI clients running on UNIX, Linux, and Windows systems. It specifies the location of the key database file in which keys and certificates are stored. The key database file must have a file name of the form `zzz.kdb`, where `zzz` is user-selectable. The *KeyRepository* field contains the path to this file, along with the file name stem (all characters in the file name up to but not including the final `.kdb`). The `.kdb` file suffix is added automatically.

Each key database file has an associated *password stash file*. This holds encoded passwords that are used to allow programmatic access to the key database. The password stash file must reside in the same directory and have the same file stem as the key database, and must end with the suffix `.sth`.

For example, if the *KeyRepository* field has the value `/xxx/yyy/key`, the key database file must be `/xxx/yyy/key.kdb`, and the password stash file must be `/xxx/yyy/key.sth`, where `xxx` and `yyy` represent directory names.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. The value is not checked; if there is an error in accessing the key repository, the call fails with reason code MQRC_KEY_REPOSITORY_ERROR.

To run an SSL connection from an IBM MQ MQI client, set *KeyRepository* to a valid key database file name.

This is an input field. The length of this field is given by MQ_SSL_KEY_REPOSITORY_LENGTH. The initial value of this field is the null string in C, and blank characters in other programming languages.

KeyResetCount (MQLONG)

This represents the total number of unencrypted bytes sent and received within an SSL or TLS conversation before the secret key is renegotiated.

The number of bytes includes control information sent by the MCA.

If you specify an SSL or TLS secret key reset count in the range 1 byte through 32 KB, SSL or TLS channels will use a secret key reset count of 32 KB. This is to avoid the processing cost of excessive key resets which would occur for small SSL or TLS secret key reset values.

This is an input field. The value is a number in the range 0 through 999 999 999, with a default value of 0. Use a value of 0 to indicate that secret keys are never renegotiated.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQSCO_STRUC_ID

Identifier for SSL configuration options structure.

For the C programming language, the constant MQSCO_STRUC_ID_ARRAY is also defined; this has the same value as MQSCO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQSCO_STRUC_ID.

Version (MQLONG)

This is the structure version number; the value must be:

MQSCO_VERSION_1

Version-1 SSL configuration options structure.

MQSCO_VERSION_2

Version-2 SSL configuration options structure.

MQSCO_VERSION_3

Version-3 SSL configuration options structure.

MQSCO_VERSION_4

Version-4 SSL configuration options structure.

MQSCO_VERSION_5

Version-5 SSL configuration options structure.

The following constant specifies the version number of the current version:

MQSCO_CURRENT_VERSION

Current version of SSL configuration options structure.

This is always an input field. The initial value of this field is MQSCO_VERSION_1.

Initial values and language declarations for MQSCO

Table 534. Initial values of fields in MQSCO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQSCO_STRUC_ID	'SC0~'
<i>Version</i>	MQSCO_CURRENT_VERSION	1
<i>KeyRepository</i>	None	Null string or blanks

Table 534. Initial values of fields in MQSCO (continued)

Field name	Name of constant	Value of constant
<i>CryptoHardware</i>	None	Null string or blanks
<i>AuthInfoRecCount</i>	None	0
<i>AuthInfoRecOffset</i>	None	0
<i>AuthInfoRecPtr</i>	None	Null pointer or null bytes
<i>KeyResetCount</i>	MQSCO_RESET_COUNT_DEFAULT	0
<i>FipsRequired</i>	MQSSL_FIPS_NO	0
<i>EncryptionPolicySuiteB</i>	MQ_SUITE_B_NONE, MQ_SUITE_B_NOT_AVAILABLE, MQ_SUITE_B_NOT_AVAILABLE, MQ_SUITE_B_NOT_AVAILABLE	1, 0, 0, 0
<i>CertificateValPolicy</i>	MQ_CERT_VAL_POLICY_DEFAULT	0

Notes:

1. The symbol – represents a single blank character.
2. In the C programming language, the macro variable MQSCO_DEFAULT contains the values listed in the table. Use it in the following way to provide initial values for the fields in the structure:

```
MQSCO MySCO = {MQSCO_DEFAULT};
```

C declaration

```
typedef struct tagMQSCO MQSCO;
struct tagMQSCO {
    MQCHAR4    StrucId;                /* Structure identifier */
    MQLONG     Version;                /* Structure version number */
    MQCHAR256   KeyRepository;         /* Location of SSL key */
                                          /* repository */
    MQCHAR256   CryptoHardware;        /* Cryptographic hardware */
                                          /* configuration string */
    MQLONG      AuthInfoRecCount;      /* Number of MQAIR records */
                                          /* present */
    MQLONG      AuthInfoRecOffset;     /* Offset of first MQAIR */
                                          /* record from start of */
                                          /* MQSCO structure */
    PMQAIR      AuthInfoRecPtr;        /* Address of first MQAIR */
                                          /* record */
    /* Ver:1 */
    MQLONG      KeyResetCount;         /* Number of unencrypted */
                                          /* bytes sent/received */
                                          /* before secret key is */
                                          /* reset */
    MQLONG      FipsRequired;          /* Using FIPS-certified */
    /* Ver:2 */
                                          /* algorithms */
    MQLONG      EncryptionPolicySuiteB[4]; /* Use only Suite B */
    /* Ver:3 */
    MQLONG      CertificateValPolicy;   /* cryptographic algorithms */
                                          /* Certificate validation */
                                          /* policy */
    /* Ver:4 */
    MQCHAR64    CertificateLabel;      /* Certificate label */
    /* Ver:5 */
}
```

COBOL declaration

```
**  MQSCO structure
10 MQSCO.
```

```

**      Structure identifier
15 MQSCO-STRUCID          PIC X(4).
**      Structure version number
15 MQSCO-VERSION          PIC S9(9) BINARY.
**      Location of SSL key repository
15 MQSCO-KEYREPOSITORY    PIC X(256).
**      Cryptographic hardware configuration string
15 MQSCO-CRYPTOHardware   PIC X(256).
**      Number of MQAIR records present
15 MQSCO-AUTHINFORECCOUNT PIC S9(9) BINARY.
**      Offset of first MQAIR record from start of MQSCO structure
15 MQSCO-AUTHINFORECOFFSET PIC S9(9) BINARY.
**      Address of first MQAIR record
15 MQSCO-AUTHINFORECPtr   POINTER.
**      Version 1 **
**      Number of unencrypted bytes sent/received before secret key is
**      reset
15 MQSCO-KEYRESETCOUNT PIC S9(9) BINARY.
**      Using FIPS-certified algorithms
15 MQSCO-FIPSREQUIRED PIC S9(9) BINARY.
**      Version 2 **
**      Use only Suite B cryptographic algorithms
15 MQSCO-ENCRYPTIONPOLICYSUITEB PIC S9(9) BINARY OCCURS 4.
**      Version 3 **
**      Certificate validation policy setting
15 MQSCO-CERTIFICATEVALPOLICY PIC S9(9) BINARY.
**      Version 4

```

PL/I declaration

```

dcl
  1 MQSCO based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version number */
    3 KeyRepository    char(256),        /* Location of SSL key
                                          repository */
    3 CryptoHardware   char(256),        /* Cryptographic hardware
                                          configuration string */
    3 AuthInfoRecCount fixed bin(31),    /* Number of MQAIR records
                                          present */
    3 AuthInfoRecOffset fixed bin(31),   /* Offset of first MQAIR record
                                          from start of MQSCO structure */
    3 AuthInfoRecPtr   pointer,          /* Address of first MQAIR record */
    3 KeyResetCount    fixed bin(31),    /* Key reset count */
/* Version 1 */
    3 FipsRequired     fixed bin(31),    /* FIPS required */
/* Version 2 */
    3 EncryptionPolicySuiteB (4) fixed bin(31), /* Suite B encryption policy */
/* Version 3 */
    3 CertificateValPolicy fixed bin(31); /* Certificate validation policy */
/* Version 4 */

```

Visual Basic declaration

```

Type MQSCO
  StrucId      As String*4    'Structure identifier'
  Version      As Long        'Structure version number'
  KeyRepository As String*256 'Location of SSL key repository'
  CryptoHardware As String*256 'Cryptographic hardware configuration'
                                'string'
  AuthInfoRecCount As Long    'Number of MQAIR records present'
  AuthInfoRecOffset As Long   'Offset of first MQAIR record from'
                                'start of MQSCO structure'
  AuthInfoRecPtr  As MQPTR    'Address of first MQAIR record'
  KeyResetCount  As Long      'Number of unencrypted bytes sent/received before secret key
is reset'
  'Version 1'
  FipsRequired   As Long      'Mandatory FIPS CipherSpecs?'
  'Version 2'
End Type

```

MQSD - Subscription descriptor

The following table summarizes the fields in the structure.

Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options
<i>ObjectName</i>	Object name	ObjectName
<i>AlternateUserId</i>	Alternate User Id	AlternateUserId
<i>AlternateSecurityId</i>	Alternate Security ID	AlternateSecurityId
<i>SubExpiry</i>	Subscription Expiry	SubExpiry
<i>ObjectString</i>	Object String	ObjectString
<i>SubName</i>	Subscription Name	SubName
<i>SubUserData</i>	Subscription user data	SubUserData
<i>SubCorrelId</i>	Subscription Correlation ID	SubCorrelId
<i>PubPriority</i>	Publication priority	PubPriority
<i>PubAccountingToken</i>	Publication Accounting Token	PubAccountingToken
<i>PubAppIdentityData</i>	Publication application identity data	PubAppIdentityData
<i>SelectionString</i>	String providing selection criteria	SelectionString
<i>SubLevel</i>	Subscription Level	SubLevel
<i>ResObjectString</i>	Long object name	ResObjectString

Overview for MQSD

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS, plus IBM MQ MQI clients connected to these systems.

Purpose: The MQSD structure is used to specify details about the subscription being made.

The structure is an input/output parameter on the MQSUB call. For more information, see [MQSUB usage notes](#).

Managed subscriptions: If an application has no specific need to use a particular queue as the destination for those publications that match its subscription, it can use the managed subscription feature. If an application elects to use a managed subscription, the queue manager informs the subscriber about the destination where published messages are sent, by providing an object handle as an output from the MQSUB call. For more information, see [Hobj \(MQHOBJ\) - input/output](#).

When the subscription is removed, the queue manager also undertakes to clean up messages that have not been retrieved from the managed destination, in the following situations:

- When the subscription is removed - by use of MQCLOSE with MQCO_REMOVE_SUB - and the managed Hobj is closed.
- By implicit means when the connection is lost to an application using a non-durable subscription (MQSO_NON_DURABLE)
- By expiration when a subscription is removed because it has expired and the managed Hobj is closed.

You must use managed subscriptions with non-durable subscriptions, so that this clean up can occur, and so that messages for closed non-durable subscriptions do not take up space in your queue manager. Durable subscriptions can also use managed destinations.

Version: The current version of MQSD is MQSD_VERSION_1.

Character set and encoding: Data in MQSD must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQSD

The MQSD structure contains the following fields; the fields are described in alphabetical order:

AlternateSecurityId (MQBYTE40)

This is a security identifier that is passed with the *AlternateUserId* to the authorization service to allow appropriate authorization checks to be performed.

AlternateSecurityId is used only if MQSO_ALTERNATE_USER_AUTHORITY is specified, and the *AlternateUserId* field is not entirely blank up to the first null character or the end of the field.

On return from an MQSUB call using MQSO_RESUME, this field is unchanged.

See the description of [“AlternateSecurityId \(MQBYTE40\)” on page 457](#) in the MQOD data type for more information.

AlternateUserId (MQCHAR12)

If you specify MQSO_ALTERNATE_USER_AUTHORITY, this field contains an alternative user identifier that is used to check the authorization for the subscription and for output to the destination queue (specified in the *Hobj* parameter of the MQSUB call), in place of the user identifier that the application is currently running under.

If successful, the user identifier specified in this field is recorded as the subscription owning user identifier in place of the user identifier that the application is currently running under.

If MQSO_ALTERNATE_USER_AUTHORITY is specified and this field is entirely blank up to the first null character or the end of the field, the subscription can succeed only if no user authorization is needed to subscribe to this topic with the options specified or the destination queue for output.

If MQSO_ALTERNATE_USER_AUTHORITY is not specified, this field is ignored.

The following differences exist in the environments indicated:

- On z/OS, only the first 8 characters of *AlternateUserId* are used to check the authorization for the subscription. However, the current user identifier must be authorized to specify this particular alternative user identifier; all 12 characters of the alternative user identifier are used for this check. The user identifier must contain only characters allowed by the external security manager.

On return from an MQSUB call using MQSO_RESUME, this field is unchanged.

This is an input field. The length of this field is given by MQ_USER_ID_LENGTH. The initial value of this field is the null string in C, and 12 blank characters in other programming languages.

ObjectName (MQCHAR48)

This is the name of the topic object as defined on the local queue manager.

The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)

- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but can contain trailing blanks. Use a null character to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply in the environments indicated:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- On z/OS:
 - Avoid names that begin or end with an underscore; they cannot be processed by the operations and control panels.
 - The percent character has a special meaning to RACF. If RACF is used as the external security manager, names must not contain the percent. If they do, those names are not included in any security checks when RACF generic profiles are used.
- On IBM i, names containing lowercase characters, forward slash, or percent, must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified for names that occur as fields in structures or as parameters on calls.

The *ObjectName* is used to form the full topic name.

The full topic name can be built from two different fields: *ObjectName* and *ObjectString*. For details of how these two fields are used, see [“Using topic strings” on page 554](#).

If the object identified by the *ObjectName* field cannot be found, the call fails with reason code MQRC_UNKNOWN_OBJECT_NAME even if there is a string specified in *ObjectString*.

On return from an MQSUB call using the MQSO_RESUME option this field is unchanged.

The length of this field is given by MQ_TOPIC_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

If altering an existing subscription using the MQSO_ALTER option, the name of the topic object subscribed to cannot be changed. This field and the *ObjectString* field can be omitted. If they are provided, they must resolve to the same full topic name. If they do not, the call fails with MQRC_TOPIC_NOT_ALTERABLE.

ObjectString (MQCHARV)

This is the long object name to be used.

The *ObjectString* is used to form the Full topic name.

The full topic name can be built from two different fields: *ObjectName* and *ObjectString*. For details of how these two fields are used, see [“Using topic strings” on page 554](#).

The maximum length of *ObjectString* is 10240.

If *ObjectString* is not specified correctly, according to the description of how to use the MQCHARV structure, or if it exceeds the maximum length, the call fails with reason code MQRC_OBJECT_STRING_ERROR.

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

If there are wildcards in the *ObjectString* the interpretation of those wildcards can be controlled using the Wildcard options specified in the Options field of the MQSD.

On return from an MQSUB call using the MQSO_RESUME option this field is unchanged. The full topic name used is returned in the *ResObjectString* field if a buffer is provided.

If altering an existing subscription using the MQSO_ALTER option, the long name of the topic object subscribed to cannot be changed. This field and the *ObjectName* field can be omitted. If they are provided they must resolve to the same full topic name or the call fails with MQRC_TOPIC_NOT_ALTERABLE.

Options (MQLONG)

This provides options to control the action of the MQSUB call.

You must specify at least one of the following options:

- MQSO_ALTER
- MQSO_RESUME
- MQSO_CREATE

To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

Combinations that are not valid are noted in this topic; any other combinations are valid.

Access or creation options: Access and creation options control whether a subscription is created, or whether an existing subscription is returned or altered. You must specify at least one of these options. The table displays valid combinations of access and creation options.

Combination of options	Notes
MQSO_CREATE	Creates a subscription if one does not exist. This combination fails if the subscription exists.
MQSO_RESUME	Resumes an existing subscription. This combination fails if no subscription exists.
MQSO_CREATE + MQSO_RESUME	Creates a subscription if one does not exist and resumes a matching one, if it does exist. This combination is useful when it is used in an application that is run a number of times.
MQSO_ALTER (see note)	Resumes an existing subscription, altering any fields to match that specified in the MQSD. This combination fails if no subscription exists.
MQSO_CREATE + MQSO_ALTER (see note)	Creates a subscription if one does not exist and resumes a matching one, if it does exist, altering any fields to match that specified in the MQSD. This combination is useful combination when used in an application that wants to ensure that its subscription is in a certain state before proceeding.

Note:

Options specifying MQSO_ALTER can also specify MQSO_RESUME, but this combination has no additional effect to specifying MQSO_ALTER alone. MQSO_ALTER implies MQSO_RESUME, because calling MQSUB to alter a subscription implies that the subscription will also be resumed. The opposite is not true, however: resuming a subscription does not imply it is to be altered.

MQSO_CREATE

Create a new subscription for the topic specified. If a subscription using the same *SubName* exists, the call fails with MQRC_SUB_ALREADY_EXISTS. This failure can be avoided by combining the MQSO_CREATE option with MQSO_RESUME. The *SubName* is not always necessary. For more details, see the description of that field.

Combining MQSO_CREATE with MQSO_RESUME returns a handle to a pre-existing subscription for the specified *SubName* if one is found; if there is no existing subscription, a new one is created using all the fields provided in the MQSD.

MQSO_CREATE can also be combined with MQSO_ALTER to similar effect.

MQSO_RESUME

Return a handle to a pre-existing subscription which matches that specified by *SubName*. No changes are made to the matching subscriptions attributes and they are returned on output in the MQSD structure. Only the following MQSD fields are used: StrucId, Version, Options, AlternateUserId and AlternateSecurityId, and SubName.

The call fails with reason code MQRC_NO_SUBSCRIPTION if a subscription does not exist matching the full subscription name. This failure can be avoided by combining the MQSO_CREATE option with MQSO_RESUME.

The user ID of the subscription is the user ID that created the subscription, or if it has been later altered by a different user ID, it is the user ID of the most recent successful alteration. If an AlternateUserId is used, and use of alternate user IDs is allowed for that user, the alternate user ID is recorded as the user ID that created the subscription instead of the user ID under which the subscription was made.

If a matching subscription exists that was created without the MQSO_ANY_USERID option, and the user ID of the subscription is different from that of the application requesting a handle to the subscription, the call fails with reason code MQRC_IDENTITY_MISMATCH.

If a matching subscription exists and is currently in use, the call fails with MQRC_SUBSCRIPTION_IN_USE.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call fails with MQRC_INVALID_SUBSCRIPTION.

MQSO_RESUME is implied by MQSO_ALTER so you do not need to combine it with that option. However, combining the two options does not cause an error.

MQSO_ALTER

Return a handle to a pre-existing subscription with the full subscription name matching that specified by the name in *SubName*. Any attributes of the subscription that are different from that specified in the MQSD are altered in the subscription unless alteration is disallowed for that attribute. Details are noted in the description of each attribute and are summarized in the following table. If you try to alter an attribute that cannot be changed, or to alter a subscription that has set the MQSO_IMMUTABLE option, the call fails with the reason code shown in the following table.

The call fails with reason code MQRC_NO_SUBSCRIPTION if a subscription matching the full subscription name does not exist. You can avoid this failure by combining the MQSO_CREATE option with MQSO_ALTER.

Combining MQSO_CREATE with MQSO_ALTER returns a handle to a pre-existing subscription for the specified *SubName* if one is found; if there is no existing subscription, a new one is created using all the fields provided in the MQSD.

The user ID of the subscription is the user ID that created the subscription, or if it is later altered by a different user ID, it is the user ID of the most recent, successful alteration. If an AlternateUserId is used, and use of alternate user IDs is allowed for that user, then the alternate user ID is recorded as the user ID that created the subscription instead of the user ID under which the subscription was made.

If a matching subscription exists that was created without the option MQSO_ANY_USERID and the user ID of the subscription is different from that of the application requesting a handle to the subscription, the call fails with reason code MQRC_IDENTITY_MISMATCH.

If a matching subscription exists and is currently in use, the call fails with MQRC_SUBSCRIPTION_IN_USE.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call fails with MQRC_INVALID_SUBSCRIPTION.

The following table shows the ability of MQSO_ALTER to alter attribute values in MQSD and MQSUB.

Table 535. Attributes in MQSD and MQSUB that can be altered

Data type descriptor or function call	Field name	Can this attribute be altered using MQSO ALTER	Reason Code
MQSD	Durability options	No	MQRC_DURABILITY_NOT_ALTERABLE
MQSD	Destination Options	Yes	None
MQSD	Registration options	Yes (see note “1” on page 544)	MQRC_GROUPING_NOT_ALTERABLE if you try to alter MQSO_GROUP_SUB
MQSD	Publication options	Yes (see note “2” on page 544)	None
MQSD	Wildcard options	No	MQRC_TOPIC_NOT_ALTERABLE
MQSD	Other options	No (see note “3” on page 544)	None
MQSD	ObjectName	No	MQRC_TOPIC_NOT_ALTERABLE
MQSD	AlternateUserId	No (see note “4” on page 544)	None
MQSD	AlternateSecurityId	No (see note “4” on page 544)	None
MQSD	SubExpiry	Yes	None
MQSD	ObjectString	No	MQRC_TOPIC_NOT_ALTERABLE
MQSD	SubName	No (see note “5” on page 544)	None
MQSD	SubUserData	Yes	None
MQSD	SubCorrelId	Yes (see note “6” on page 544)	MQRC_GROUPING_NOT_ALTERABLE when in a grouped subscription
MQSD	PubPriority	Yes	None
MQSD	PubAccountingToken	Yes	None
MQSD	PubApplIdentityData	Yes	None
MQSD	SubLevel	No	MQRC_SUBLEVEL_NOT_ALTERABLE
MQSUB	Hobj	Yes (see note “6” on page 544)	MQRC_GROUPING_NOT_ALTERABLE when in a grouped subscription

Notes:

1. MQSO_GROUP_SUB cannot be altered.
2. MQSO_NEW_PUBLICATIONS_ONLY cannot be altered because it is not part of the subscription
3. These options are not part of the subscription
4. This attribute is not part of the subscription
5. This attribute is the identity of the subscription being altered
6. Alterable except when part of a grouped sub (MQSO_GROUP_SUB)

Durability options: The following options control how durable the subscription is. You can specify only one of these options. If you are altering an existing subscription using the MQSO ALTER option, you cannot change the durability of the subscription. On return from an MQSUB call using MQSO_RESUME, the appropriate durability option is set.

MQSO_DURABLE

Request that the subscription to this topic remains until it is explicitly removed using MQCLOSE with the MQCO_REMOVE_SUB option. If this subscription is not explicitly removed it will remain even after this applications connection to the queue manager is closed.

If a durable subscription is requested to a topic that is defined as not allowing durable subscriptions, the call fails with MQRC_DURABILITY_NOT_ALLOWED.

MQSO_NON_DURABLE

Request that the subscription to this topic is removed when the applications connection to the queue manager is closed, if it is not already explicitly removed. MQSO_NON_DURABLE is the opposite of the MQSO_DURABLE option, and is defined to aid program documentation. It is the default if neither is specified.

Destination options: The following option controls the destination that publications for a topic that has been subscribed to are sent to. If altering an existing subscription using the MQSO_ALTER option, the destination used for publications for the subscription can be changed. On return from an MQSUB call using MQSO_RESUME, this option is set if appropriate.

MQSO_MANAGED

Request that the destination that the publications are sent to is managed by the queue manager.

The object handle returned in *Hobj* represents a queue manager managed queue and is for use with subsequent MQGET, MQCB, MQINQ, or MQCLOSE calls.

An object handle returned from a previous MQSUB call cannot be provided in the *Hobj* parameter when MQSO_MANAGED is not specified.

MQSO_NO_MULTICAST

Request that the destination that the publications are sent to is not a multicast group address. This option is only valid when combined with the MQSO_MANAGED option. When a handle to a queue is provided in the *Hobj* parameter, multicast cannot be used for this subscription, and the option is not valid.

If the topic is defined to only allow multicast subscriptions, using the MCAST (ONLY) setting, then the call fails with reason code MQRC_MULTICAST_REQUIRED.

Scope Option: The following option controls the scope of the subscription being made. If altering an existing subscription using the MQSO_ALTER option, this subscription scope option cannot be changed. On returning from an MQSUB call using MQSO_RESUME, the appropriate scope option is set.

MQSO_SCOPE_QMGR

This subscription is made only on the local queue manager. No proxy subscription is distributed to other queue managers in the network. Only publications that are published at this queue manager are sent to this subscriber. This overrides any behavior set using the SUBSCOPE topic attribute.

Note: If not set, the subscription scope is determined by the SUBSCOPE topic attribute.

Registration options: The following options control the details of the registration that is made to the queue manager for this subscription. If altering an existing subscription using the MQSO_ALTER option, these registration options can be changed. On return from an MQSUB call using MQSO_RESUME the appropriate registration options is set.

MQSO_GROUP_SUB

This subscription is to be grouped with other subscriptions of the same SubLevel using the same queue and specifying the same correlation ID so that any publications to topics that would cause more than one publication message to be provided to the group of subscriptions, due to an overlapping set of topic strings being used, only causes one message to be delivered to the queue. If this option is not used, then each unique subscription (identified by SubName) that matches is provided with a copy of the publication which could mean more than one copy of the publication may be placed on the queue shared by a number of subscriptions.

Only the most significant subscription in the group is provided with a copy of the publication. The most significant subscription is based on the Full topic name up to the point where a wildcard is found. If a mixture of wildcard schemes is used within the group, only the position of the wildcard is important. You are advised not to combine different wildcard schemes within a group of subscriptions that share the same queue.

When creating a new grouped subscription it must still have a unique SubName, but if it matches the full topic name of an existing subscription in the group, the call fails with MQRC_DUPLICATE_GROUP_SUB.

If the most significant subscription in group also specifies MQSO_NOT_OWN_PUBS and this is a publication from the same application, then no publication is delivered to the queue.

When altering a subscription made with this option, the fields which imply the grouping, Hobj on the MQSUB call (representing the queue and queue manager name), and the SubCorrelId cannot be changed. Attempting to alter them causes the call to fail with MQRC_GROUPING_NOT_ALTERABLE.

This option must be combined with MQSO_SET_CORREL_ID with a SubCorrelId that is not set to MQCI_NONE, and cannot be combined with MQSO_MANAGED.

MQSO_ANY_USERID

When MQSO_ANY_USERID is specified, the identity of the subscriber is not restricted to a single user ID. This allows any user to alter or resume the subscription when they have suitable authority. Only a single user may have the subscription at any one time. An attempt to resume use of a subscription currently in use by another application causes the call to fail with MQRC_SUBSCRIPTION_IN_USE.

To add this option to an existing subscription the MQSUB call (using MQSO_ALTER) must come from the same user ID as the original subscription itself.

If an MQSUB call refers to an existing subscription with MQSO_ANY_USERID set, and the user ID differs from the original subscription, the call succeeds only if the new user ID has authority to subscribe to the topic. On successful completion, future publications to this subscriber are put to the subscribers queue with the new user ID set in the publication message.

Do not specify both MQSO_ANY_USERID and MQSO_FIXED_USERID. If neither is specified, the default is MQSO_FIXED_USERID.

MQSO_FIXED_USERID

When MQSO_FIXED_USERID is specified, the subscription can be altered or resumed by only the last user ID to alter the subscription. If the subscription has not been altered, it is the user ID that created the subscription.

If an MQSUB verb refers to an existing subscription with MQSO_ANY_USERID set and alters the subscription using MQSO_ALTER to use option MQSO_FIXED_USERID, the user ID of the subscription is now fixed at this new user ID. The call succeeds only if the new user ID has authority to subscribe to the topic.

If a user ID other than the one recorded as owning a subscription tries to resume or alter an MQSO_FIXED_USERID subscription, the call fails with MQRC_IDENTITY_MISMATCH. The owning user ID of a subscription can be viewed using the DISPLAY SBSTATUS command.

Do not specify both MQSO_ANY_USERID and MQSO_FIXED_USERID. If neither is specified, the default is MQSO_FIXED_USERID.

Publication options: The following options control the way publications are sent to this subscriber. If altering an existing subscription using the MQSO_ALTER option, these publication options can be changed.

MQSO_NOT_OWN_PUBS

Tells the broker that the application does not want to see any of its own publications. Publications are considered to originate from the same application if the connection handles are the same. On return from an MQSUB call using MQSO_RESUME, this option is set if appropriate.

MQSO_NEW_PUBLICATIONS_ONLY

No currently retained publications are to be sent, when this subscription is created, only new publications. This option only applies when MQSO_CREATE is specified. Any subsequent changes to a subscription do not alter the flow of publications and so any publications retained on a topic, will have already been sent to the subscriber as new publications.

If this option is specified without MQSO_CREATE the call fails with MQRC_OPTIONS_ERROR. On return from an MQSUB call using MQSO_RESUME, this option is not set even if the subscription was created using this option.

If this option is not used, previously retained messages are sent to the destination queue provided. If this action fails due to an error, either MQRC_RETAINED_MSG_Q_ERROR or MQRC_RETAINED_NOT_DELIVERED, the creation of the subscription fails.

MQSO_PUBLICATIONS_ON_REQUEST

Setting this option indicates that the subscriber will request information specifically when required. The queue manager does not send unsolicited messages to the subscriber. The retained publication (or possibly multiple publications if a wildcard is specified in the topic) is sent to the subscriber each time an MQSUBRQ call is made using the Hsub handle from a previous MQSUB call. No publications are sent as a result of the MQSUB call using this option. On return from an MQSUB call using MQSO_RESUME, this option is set if appropriate.

This option is not valid in combination with a SubLevel greater than 1.

Read ahead options: The following options control whether non-persistent messages are sent to an application ahead of the application requesting them.

MQSO_READ_AHEAD_AS_Q_DEF

If the MQSUB call uses a managed handle, the default read ahead attribute of the model queue associated with the topic subscribed to determines whether messages are sent to the application before the application requests them.

This is the default value.

MQSO_NO_READ_AHEAD

If the MQSUB call uses a managed handle, messages are not sent to the application before the application requests them.

MQSO_READ_AHEAD

If the MQSUB call uses a managed handle, messages might be sent to the application before the application requests them.

Note:

The following notes apply to the read ahead options:

1. Only one of these options can be specified. If both MQOO_READ_AHEAD and MQOO_NO_READ_AHEAD are specified, reason code MQRC_OPTIONS_ERROR is returned. These options are only applicable if MQSO_MANAGED is specified.
2. They are not applicable for MQSUB when a queue is passed which has been opened previously. Read ahead might not be enabled when requested. The MQGET options used on the first MQGET call might prevent read ahead from being enabled. Also, read ahead is disabled when the client is connecting to a queue manager where read ahead is not supported. If the application is not running as an IBM MQ client, these options are ignored.

Wildcard options: The following options control how wildcards are interpreted in the string provided in the ObjectString field of the MQSD. You can specify only one of these options. If altering an existing subscription using the MQSO_ALTER option, these wildcard options cannot be changed. On return from an MQSUB call using MQSO_RESUME, the appropriate wildcard option is set.

MQSO_WILDCARD_CHAR

Wildcards only operate on characters within the topic string.

The behavior defined by MQSO_WILDCARD_CHAR is shown in the following table.

Special Character	Behavior
Forward slash (/)	No significance, just another character
Asterisk (*)	Wildcard, zero or more characters
Question mark (?)	Wildcard, 1 character

Special Character	Behavior
Percent sign (%)	Escape character to allow the characters (*), (?) or (%) to be used in a string and not be interpreted as a special character, for example, (%*), (%?) or (%%).

For example, publishing on the following topic:

```
/level0/level1/level2/level3/level4
```

matches subscribers using the following topics:

```
*
/*
/ level0/level1/level2/level3/*
/ level0/level1/*/*level3/level4
/ level0/level1/le?e12/level3/level4
```

Note: This use of wildcards supplies exactly the meaning provided in IBM MQ V6 and WebSphere MB V6 when using MQRFH1 formatted messages for publish/subscribe. It is recommended that this is not used for newly written applications and is only used for applications that were previously running against that version and have not been changed to use the default wildcard behavior as described in MQSO_WILDCARD_TOPIC.

MQSO_WILDCARD_TOPIC

Wildcards only operate on topic elements within the topic string. This is the default behavior if none is chosen.

The behavior required by MQSO_WILDCARD_TOPIC is shown in the following table:

Special Character	Behavior
(/)	Topic level separator
Number sign (#)	Wildcard: multiple topic level
Plus sign (+)	Wildcard: single topic level

Notes:

The (+) and (#) are not treated as wildcards if they are mixed in with other characters (including themselves) within a topic level. In the following string, the (#) and (+) characters are treated as ordinary characters.

```
level0/level1/#+/level3/level#
```

For example, publishing on the following topic:

```
/level0/level1/level2/level3/level4
```

matches subscribers using the following topics:

```
#
/#
/ level0/level1/level2/level3/#
/ level0/level1/+/*level3/level4
```

Other options: The following options control the way the API call is issued rather than the subscription. On return from an MQSUB call using MQSO_RESUME, these options are unchanged. See [“AlternateUserId \(MQCHAR12\)”](#) on page 540 for more details.

MQSO_ALTERNATE_USER_AUTHORITY

The *AlternateUserId* field contains a user identifier to use to validate this MQSUB call. The call can succeed only if this *AlternateUserId* is authorized to open the object with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so.

MQSO_SET_CORREL_ID

The subscription is to use the correlation identifier supplied in the *SubCorrelId* field. If this option is not specified, a correlation identifier is automatically created by the queue manager at subscription time and is returned to the application in the *SubCorrelId* field. For more information, see [“SubCorrelId \(MQBYTE24\)”](#) on page 551 for more information.

This option cannot be combined with MQSO_MANAGED.

MQSO_SET_IDENTITY_CONTEXT

The subscription is to use the accounting token and application identity data supplied in the *PubAccountingToken* and *PubApplIdentityData* fields.

If this option is specified, the same authorization check is carried out as if the destination queue was accessed using an MQOPEN call with MQOO_SET_IDENTITY_CONTEXT, except in the case where the MQSO_MANAGED option is also used in which case there is no authorization check on the destination queue.

If this option is not specified, the publications sent to this subscriber have default context information associated with them as follows:

Field in MQMD	Value used
<i>UserIdentifier</i>	The user ID associated with the subscription at the time the subscription was made.
<i>AccountingToken</i>	Determined from the environment if possible; Set to MQACT_NONE if not.
<i>ApplIdentityData</i>	Set to blanks

This option is only valid with MQSO_CREATE and MQSO_ALTER. If used with MQSO_RESUME, the *PubAccountingToken* and *PubApplIdentityData* fields are ignored, so this option has no effect.

If a subscription is altered without using this option where previously the subscription supplied identity context information, default context information is generated for the altered subscription.

If a subscription allowing different user IDs to use it with option MQSO_ANY_USERID, is resumed by a different user ID, default identity context is generated for the new user ID now owning the subscription and any subsequent publications are delivered containing the new identity context.

MQSO_FAIL_IF_QUIESCING

The MQSUB call fails if the queue manager is in quiescing state. On z/OS, for a CICS or IMS application, this option also forces the MQSUB call to fail if the connection is in quiescing state.

PubAccountingToken (MQBYTE32)

This is the value that will be in the *AccountingToken* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. *AccountingToken* is part of the identity context of the message. For more information about message context, see [Message context](#). For more information about the *AccountingToken* field in the MQMD, see [“AccountingToken \(MQBYTE32\)”](#) on page 400

You can use the following special value for the *PubAccountingToken* field:

MQACT_NONE

No accounting token is specified.

The value is binary zero for the length of the field.

For the C programming language, the constant `MQACT_NONE_ARRAY` is also defined; this has the same value as `MQACT_NONE`, but is an array of characters instead of a string.

If the option `MQSO_SET_IDENTITY_CONTEXT` is not specified, the accounting token is generated by the queue manager as default context information and this field is an output field which contains the *AccountingToken* which will be set in each message published for this subscription.

If the option `MQSO_SET_IDENTITY_CONTEXT` is specified, the accounting token is being generated by the user and this field is an input field which contains the *AccountingToken* to be set in each publication for this subscription.

The length of this field is given by `MQ_ACCOUNTING_TOKEN_LENGTH`. The initial value of this field is `MQACT_NONE`.

If altering an existing subscription using the `MQSO_ALTER` option, the value of *AccountingToken* in any future publication messages can be changed.

On return from an `MQSUB` call using `MQSO_RESUME`, this field is set to the current *AccountingToken* being used for the subscription.

PubApplIdentityData (MQCHAR32)

This is the value that is in the *ApplIdentityData* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. *ApplIdentityData* is part of the identity context of the message. For more information about message context, see [Message context](#). For more information about the *ApplIdentityData* field in the MQMD, see [“ApplIdentityData \(MQCHAR32\)” on page 401](#)

If the option `MQSO_SET_IDENTITY_CONTEXT` is not specified, the *ApplIdentityData* which is set in each message published for this subscription is blanks, as default context information.

If the option `MQSO_SET_IDENTITY_CONTEXT` is specified, the *PubApplIdentityData* is being generated by the user and this field is an input field which contains the *ApplIdentityData* to be set in each publication for this subscription.

The length of this field is given by `MQ_APPL_IDENTITY_DATA_LENGTH`. The initial value of this field is the null string in C, and 32 blank characters in other programming languages.

If altering an existing subscription using the `MQSO_ALTER` option, the *ApplIdentityData* of any future publication messages can be changed.

On return from an `MQSUB` call using `MQSO_RESUME`, this field is set to the current *ApplIdentityData* being used for the subscription.

PubPriority (MQLONG)

This is the value that will be in the *Priority* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. For more information about the *Priority* field in the MQMD, see [“Priority \(MQLONG\)” on page 426](#).

The value must be greater than or equal to zero; zero is the lowest priority. The following special values can also be used:

MQPRI_PRIORITY_AS_Q_DEF

When a subscription queue is provided in the *Hobj* field in the `MQSUB` call, and is not a managed handle, then the priority for the message is taken from the *DefPriority* attribute of this queue. If the queue is a cluster queue or there is more than one definition in the queue-name resolution path then the priority is determined when the publication message is put to the queue as described for [“Priority \(MQLONG\)” on page 426](#).

If the `MQSUB` call uses a managed handle, the priority for the message is taken from the *DefPriority* attribute of the model queue associated with the topic subscribed to.

MQPRI_PRIORITY_AS_PUBLISHED

The priority for the message is the priority of the original publication. This is the initial value of the field.

If altering an existing subscription using the MQSO_ALTER option, the *Priority* of any future publication messages can be changed.

On return from an MQSUB call using MQSO_RESUME, this field is set to the current priority being used for the subscription.

ResObjectString (MQCHARV)

This is the long object name after the queue manager resolves the name provided in *ObjectName*.

If the long object name is provided in *ObjectString* and nothing is provided in *ObjectName*, then the value returned in this field is the same as provided in *ObjectString*.

If this field is omitted (that is *ResObjectString.VSBufSize* is zero) then the *ResObjectString* is not returned, but the length is returned in *ResObjectString.VSLength*. If the length is shorter than the full *ResObjectString* then it is truncated and returns as many of the rightmost characters as can fit in the provided length.

If *ResObjectString* is specified incorrectly, according to the description of how to use the MQCHARV structure, or if it exceeds the maximum length, the call fails with reason code MQRC_RES_OBJECT_STRING_ERROR.

SelectionString (MQCHARV)

This is the string used to provide the selection criteria used when subscribing for messages from a topic.

This variable length field will be returned on output from an MQSUB call using the MQSO_RESUME option, if a buffer is provided, and also there is a positive buffer length in *VSBufSize*. If no buffer is provided on the call, only the length of the selection string will be returned in the *VSLength* field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only *VSBufSize* bytes are returned in the provided buffer.

If *SelectionString* is specified incorrectly, according to the description of how to use the “MQCHARV - Variable Length String” on page 284 structure, or if it exceeds the maximum length, the call fails with reason code MQRC_SELECTION_STRING_ERROR.

SelectionString usage is described in [Selectors](#).

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQSD_STRUC_ID

Identifier for Subscription Descriptor structure.

For the C programming language, the constant MQSD_STRUC_ID_ARRAY is also defined; this has the same value as MQSD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQSD_STRUC_ID.

SubCorrelId (MQBYTE24)

This field contains a correlation identifier common to all publications matching this subscription.



Attention: a correlation identifier can only be passed between queue managers in a publish/subscribe cluster, not a hierarchy.

All publications sent to match this subscription contain this correlation identifier in the message descriptor. If multiple subscriptions get their publications from the same queue, using MQGET by correlation identifier allows only publications for a specific subscription to be obtained. This correlation identifier can either be generated by the queue manager or by the user.

If the option MQSO_SET_CORREL_ID is not specified, the correlation identifier is generated by the queue manager and this field is an output field containing the correlation identifier that will be set in each

message published for this subscription. The generated correlation identifier consists of a 4-byte product identifier (AMQX or CSQM in either ASCII or EBCDIC) followed by a product specific implementation of a unique string.

If the option MQSO_SET_CORREL_ID is specified, the correlation identifier is generated by the user and this field is an input field containing the correlation identifier to be set in each publication for this subscription. In this case, if the field contains MQCI_NONE, the correlation identifier that is set in each message published for this subscription is the correlation identifier created by the original put of the message.

If the option MQSO_GROUP_SUB is specified and the correlation identifier specified is the same as an existing grouped subscription using the same queue and an overlapping topic string, only the most significant subscription in the group is provided with a copy of the publication.

The length of this field is given by MQ_CORREL_ID_LENGTH. The initial value of this field is MQCI_NONE.

If you are altering an existing subscription using the MQSO_ALTER option, and this field is an input field, then the subscription correlation identifier can be changed, unless the subscription is a grouped subscription, that is, it has been created using the option MQSO_GROUP_SUB, in which case the subscription correlation identifier cannot be changed.

On return from an MQSUB call using MQSO_RESUME, this field is set to the current correlation identifier for the subscription.

SubExpiry (MQLONG)

This is the time expressed in tenths of a second after which the subscription expires. No more publications will match this subscription after this interval has passed. As soon as a subscription expires, publications are no longer sent to the queue. However, the publications that are already there are not affected in any way. *SubExpiry* has no effect on publication expiry.

The following special value is recognized:

MQEI_UNLIMITED

The subscription has an unlimited expiration time.

If altering an existing subscription using the MQSO_ALTER option, the expiry of the subscription can be changed.

On return from an MQSUB call using the MQSO_RESUME option this field is set to the original expiry of the subscription and not the remaining expiry time.

SubLevel (MQLONG)

This is the level associated with the subscription. Publications are only delivered to this subscription if it is in the set of subscriptions with the highest SubLevel value less than or equal to the PubLevel used at publication time. However, if a publication has been retained, it is no longer available to subscribers at higher levels because it is republished at PubLevel 1.

The value must be in the range zero to 9. Zero is the lowest level.

The initial value of this field is 1.

For more information see [Intercepting publications](#).

If altering an existing subscription using the MQSO_ALTER option, then the SubLevel cannot be changed.

Combining a SubLevel with a value greater than 1 with the option MQSO_PUBLICATIONS_ON_REQUEST is not allowed.

On return from an MQSUB call using MQSO_RESUME, this field is set to the current level being used for the subscription.

SubUserData (MQCHARV)

This specifies the subscription user data. The data provided on the subscription in this field will be included as the MQSubUserData message property of every publication sent to this subscription.

The maximum length of *SubUserData* is 10240.

If *SubUserData* is specified incorrectly, according to the description of how to use the MQCHARV structure, or if it exceeds the maximum length, the call fails with reason code MQRC_SUB_USER_DATA_ERROR.

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

If altering an existing subscription using the MQSO_ALTER option, the subscription user data can be changed.

This variable length field is returned on output from an MQSUB call using the MQSO_RESUME option, if a buffer is provided and there is a positive buffer length in *VSBuflen*. If no buffer is provided on the call, only the length of the subscription user data is returned in the *VSLength* field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only *VSBuflen* bytes are returned in the provided buffer.

SubName (MQCHARV)

This specifies the subscription name. This field is only required if *Options* specifies the option MQSO_DURABLE, but if provided will be used by the queue manager for MQSO_NON_DURABLE as well.

If specified, *SubName* must be unique within the queue manager, because it is the method used to identify the subscription.

The maximum length of *SubName* is 10240.

This field serves two purposes. For an MQSO_DURABLE subscription, you use this field to identify a subscription so you can resume it after it has been created if you have either closed the handle to the subscription (using the MQCO_KEEP_SUB option) or have been disconnected from the queue manager. This is done using the MQSUB call with the MQSO_RESUME option. It is also displayed in the administrative view of subscriptions in the SUBNAME field in DISPLAY SBSTATUS.

If *SubName* is specified incorrectly, according to the description of how to use the MQCHARV structure, is left out when it is required (that is *SubName.VSLength* is zero), or if it exceeds the maximum length, the call fails with reason code MQRC_SUB_NAME_ERROR.

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

If altering an existing subscription using the MQSO_ALTER option, the subscription name cannot be changed, because it is the identifying field used to find the referenced subscription. It is not changed on output from an MQSUB call with the MQSO_RESUME option.

Version (MQLONG)

This is the structure version number; the value must be:

MQSD_VERSION_1

Version-1 Subscription Descriptor structure.

The following constant specifies the version number of the current version:

MQSD_CURRENT_VERSION

Current version of Subscription Descriptor structure.

This is always an input field. The initial value of this field is MQSD_VERSION_1.

Using topic strings

A topic is constructed from the subtopic identified in a topic object, and a subtopic provided by an application. You can use either subtopic as the topic name, or combine them to form a new topic name.

In an MQI program the full topic name is created by MQOPEN. It is composed of two fields used in publish/subscribe MQI calls, in the order listed:

1. The **TOPICSTR** attribute of the topic object, named in the **ObjectName** field.
2. The **ObjectString** parameter defining the subtopic provided by the application.

The resulting topic string is returned in the **ResObjectString** parameter.

These fields are considered to be present if the first character of each field is not a blank or null character, and the field length is greater than zero. If only one of the fields is present, it is used unchanged as the topic name. If neither field has a value, the call fails with reason code MQRC_UNKNOWN_OBJECT_NAME, or MQRC_TOPIC_STRING_ERROR if the full topic name is not valid.

If both fields are present, a ' / ' character is inserted between the two elements of the resultant combined topic name.

Table 536 on page 554 shows examples of topic string concatenation:

Table 536. Topic string concatenation examples			
TOPICSTR	ObjectString	Full topic name	Comment
Football/Scores	' '	Football/Scores	The TOPICSTR is used alone
' '	Football/Scores	Football/Scores	The ObjectString is used alone
Football	Scores	Football/Scores	A '/' character is added at the concatenation point
Football	/Scores	Football//Scores	An 'empty node' is produced between the two strings
/Football	Scores	/Football/Scores	The topic starts with an 'empty node'

The ' / ' character is considered as a special character, providing structure to the full topic name in [Topic trees](#), and must not be used for any other reason as the structure of the topic tree is affected. The topic " /Football " is not the same as the topic " Football ".

The following wildcard characters are special characters:

- plus sign ' + '
- number sign ' # '
- asterisk ' * '
- question mark ' ? '

These characters are not considered as invalid, however you must ensure to understand how they are used. You might prefer not to use these characters in your topic strings when publishing. Publishing on a topic string with ' # ' or ' + ' mixed in with other characters (including themselves) within a topic level, can be subscribed to with either wildcard scheme. Publishing on a topic string with ' # ' or ' + ' as the only character between two ' / ' characters produces a topic string that cannot be subscribed to explicitly by an application using the wildcard scheme MQSO_WILDCARD_TOPIC. This situation results in the application getting more publications than expected.

Example code snippet

This code snippet, extracted from the example program [Example 2: Publisher to a variable topic](#), combines a topic object with a variable topic string.

```
MQOD      td = {MQOD_DEFAULT}; /* Object Descriptor      */
td.ObjectType = MQOT_TOPIC;    /* Object is a topic      */
td.Version = MQOD_VERSION_4;   /* Descriptor needs to be V4 */
strncpy(td.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
td.ObjectString.VSPtr = topicString;
td.ObjectString.VSLength = (MQLONG)strlen(topicString);
td.ResObjectString.VSPtr = resTopicStr;
td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF QUIESCING, &Hobj, &CompCode, &Reason);
```

Initial values and language declarations for MQSD

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQSD_STRUC_ID	'SD--'
<i>Version</i>	MQSD_VERSION_1	1
<i>Options</i>	MQSO_NON_DURABLE	0
<i>ObjectName</i>	None	Null string or blanks
<i>AlternateUserId</i>	None	Null string or blanks
<i>AlternateSecurityId</i>	MQSID_NONE	Nulls
<i>SubExpiry</i>	MQEI_UNLIMITED	-1
<i>ObjectString</i>	None	Names and values as defined for MQCHARV
<i>SubName</i>	None	Names and values as defined for MQCHARV
<i>SubUserData</i>	None	Names and values as defined for MQCHARV
<i>SubCorrelId</i>	MQCI_NONE	Nulls
<i>PubPriority</i>	MQPRI_PRIORITY_AS_Q_DEF	-3
<i>PubAccountingToken</i>	MQACT_NONE	Nulls
<i>PubApplIdentityData</i>	None	Null string or blanks
<i>SelectionString</i>	None	Names and values as defined for MQCHARV
<i>SubLevel</i>	None	1
<i>ResObjectString</i>	None	Names and values as defined for MQCHARV

Field name	Name of constant	Value of constant
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQSD_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQSD MySD = {MQSD_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQSD MQSD;
struct tagMQSD {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     Options;          /* Options associated with subscribing */
    MQCHAR48   ObjectName;       /* Object name */
    MQCHAR12   AlternateUserId;   /* Alternate user identifier */
    MQBYTE40   AlternateSecurityId; /* Alternate security identifier */
    MQLONG     SubExpiry;        /* Expiry of Subscription */
    MQCHARV    ObjectString;     /* Object Long name */
    MQCHARV    SubName;          /* Subscription name */
    MQCHARV    SubUserData;      /* Subscription User data */
    MQBYTE24   SubCorrelId;      /* Correlation Id related to this subscription */
    MQLONG     PubPriority;       /* Priority set in publications */
    MQBYTE32   PubAccountingToken; /* Accounting Token set in publications */
    MQCHAR32   PubApplIdentityData; /* Appl Identity Data set in publications */
    MQCHARV    SelectionString;  /* Message selector structure */
    MQLONG     SubLevel;         /* Subscription level */
    MQCHARV    ResObjectString;  /* Resolved Long object name */
    /* Ver:1 */
};
```

COBOL declaration

```
** Address of variable length string
20 MQSD-OBJECTSTRING-VSPTR          POINTER.
** Offset of variable length string
20 MQSD-OBJECTSTRING-VSOFFSET       PIC S9(9) BINARY.
** size of buffer
20 MQSD-OBJECTSTRING-VSBUFSIZE      PIC S9(9) BINARY.
** Length of variable length string
20 MQSD-OBJECTSTRING-VSLENGTH       PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSD-OBJECTSTRING-VSCCSID        PIC S9(9) BINARY.
** Subscription name
15 MQSD-SUBNAME.
** Address of variable length string
20 MQSD-SUBNAME-VSPTR              POINTER.
** Offset of variable length string
20 MQSD-SUBNAME-VSOFFSET           PIC S9(9) BINARY.
** size of buffer
20 MQSD-SUBNAME-VSBUFSIZE          PIC S9(9) BINARY.
** Length of variable length string
20 MQSD-SUBNAME-VSLENGTH           PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSD-SUBNAME-VSCCSID            PIC S9(9) BINARY.
** Subscription User data
15 MQSD-SUBUSERDATA.
** Address of variable length string
20 MQSD-SUBUSERDATA-VSPTR          POINTER.
** Offset of variable length string
20 MQSD-SUBUSERDATA-VSOFFSET       PIC S9(9) BINARY.
** size of buffer
20 MQSD-SUBUSERDATA-VSBUFSIZE      PIC S9(9) BINARY.
** Length of variable length string
20 MQSD-SUBUSERDATA-VSLENGTH       PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSD-SUBUSERDATA-VSCCSID        PIC S9(9) BINARY.
```

```

** Correlation Id related to this subscription
15 MQSD-SUBCORRELID PIC X(24).
** Priority set in publications
15 MQSD-PUBPRIORITY PIC S9(9) BINARY.
** Accounting Token set in publications
15 MQSD-PUBACCOUNTINGTOKEN PIC X(32).
** Appl Identity Data set in publications
15 MQSD-PUBAPPLIDENTITYDATA PIC X(32).
** Message Selector
15 MQSD-SELECTIONSTRING.
** Address of variable length string
20 MQSD-SELECTIONSTRING-VSPTR POINTER.
** Offset of variable length string
20 MQSD-SELECTIONSTRING-VSOFFSET PIC S9(9) BINARY.
** size of buffer
20 MQSD-SELECTIONSTRING-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQSD-SELECTIONSTRING-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSD-SELECTIONSTRING-VSCCSID PIC S9(9) BINARY.
** Selection criteria
20 MQSD-SELECTIONSTRING-SUBLEVEL PIC S9(9) BINARY.
** Long object name
20 MQSD-SELECTIONSTRING-RESOBJSTRING PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQSD based,
3 StrucId      char(4), /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 Options      fixed bin(31), /* Options associated with subscribing */
3 ObjectName    char(48), /* Object name */
3 AlternateUserId char(12), /* Alternate user identifier */
3 AlternateSecurityId char(40), /* Alternate security identifier */
3 SubExpiry     fixed bin(31), /* Expiry of Subscription */
3 ObjectString, /* Object Long name */
5 VSPtr         pointer, /* Address of variable length string */
5 VSOffset      fixed bin(31), /* Offset of variable length string */
5 VSBufSize     fixed bin(31), /* size of buffer */
5 VSLength      fixed bin(31), /* Length of variable length string */
5 VSCCSID       fixed bin(31); /* CCSID of variable length string */
3 SubName,      /* Subscription name */
5 VSPtr         pointer, /* Address of variable length string */
5 VSOffset      fixed bin(31), /* Offset of variable length string */
5 VSBufSize     fixed bin(31), /* size of buffer */
5 VSLength      fixed bin(31), /* Length of variable length string */
5 VSCCSID       fixed bin(31); /* CCSID of variable length string */
3 SubUserData, /* Subscription User data */
5 VSPtr         pointer, /* Address of variable length string */
5 VSOffset      fixed bin(31), /* Offset of variable length string */
5 VSBufSize     fixed bin(31), /* size of buffer */
5 VSLength      fixed bin(31), /* Length of variable length string */
5 VSCCSID       fixed bin(31), /* CCSID of variable length string */
3 SubCorrelId   char(24), /* Correlation Id related to this subscription */
3 PubPriority    fixed bin(31), /* Priority set in publications */
3 PubAccountingToken char(32), /* Accounting Token set in publications */
3 PubApplIdentityData char(32), /* Appl Identity Data set in publications */
3 SelectionString, /* Message Selection */
5 VSPtr         pointer, /* Address of variable length string */
5 VSOffset      fixed bin(31), /* Offset of variable length string */
5 VSBufSize     fixed bin(31), /* size of buffer */
5 VSLength      fixed bin(31), /* Length of variable length string */
5 VSCCSID       fixed bin(31), /* CCSID of variable length string */
3 SubLevel      fixed bin(31), /* Subscription level */
3 ResObjectString, /* Resolved Long object name */
5 VSPtr         pointer, /* Address of variable length string */
5 VSOffset      fixed bin(31), /* Offset of variable length string */
5 VSBufSize     fixed bin(31), /* size of buffer */
5 VSLength      fixed bin(31), /* Length of variable length string */
5 VSCCSID       fixed bin(31); /* CCSID of variable length string */

```

High Level Assembler declaration

MQSD	DSECT		
MQSD_STRUCID	DS	CL4	Structure identifier
MQSD_VERSION	DS	F	Structure version number
MQSD-OPTIONS	DS	F	Options associated with subscribing

```

MQSD_OBJECTNAME          DS   CL48 Object name
MQSD_ALTERNATEUSERID     DS   CL12 Alternate user identifier
MQSD_ALTERNATESECURITYID DS   CL40 Alternate security identifier
MQSD_SUBEXPIRY           DS   F   Expiry of Subscription
MQSD_OBJECTSTRING        DS   0F   Object Long name
MQSD_OBJECTSTRING_VSPTR  DS   F   Address of variable length string
MQSD_OBJECTSTRING_VSOFFSET DS   F   Offset of variable length string
MQSD_OBJECTSTRING_VSBUFFSIZE DS   F   size of buffer
MQSD_OBJECTSTRING_VSLNGTH DS   F   Length of variable length string
MQSD_OBJECTSTRING_VSCCSID DS   F   CCSID of variable length string
MQSD_OBJECTSTRING_LENGTH EQU   *-MQSD_OBJECTSTRING
ORG MQSD_OBJECTSTRING
MQSD_OBJECTSTRING_AREA   DS   CL(MQSD_OBJECTSTRING_LENGTH)
*
MQSD_SUBNAME             DS   0F   Subscription name
MQSD_SUBNAME_VSPTR       DS   F   Address of variable length string
MQSD_SUBNAME_VSOFFSET    DS   F   Offset of variable length string
MQSD_SUBNAME_VSBUFFSIZE  DS   F   size of buffer
MQSD_SUBNAME_VSLNGTH     DS   F   Length of variable length string
MQSD_SUBNAME_VSCCSID     DS   F   CCSID of variable length string
MQSD_SUBNAME_LENGTH      EQU   *-MQSD_SUBNAME
ORG MQSD_SUBNAME
MQSD_SUBNAME_AREA        DS   CL(MQSD_SUBNAME_LENGTH)
*
MQSD_SUBUSERDATA         DS   0F   Subscription User data
MQSD_SUBUSERDATA_VSPTR   DS   F   Address of variable length string
MQSD_SUBUSERDATA_VSOFFSET DS   F   Offset of variable length string
MQSD_SUBUSERDATA_VSBUFFSIZE DS   F   size of buffer
MQSD_SUBUSERDATA_VSLNGTH DS   F   Length of variable length string
MQSD_SUBUSERDATA_VSCCSID DS   F   CCSID of variable length string
MQSD_SUBUSERDATA_LENGTH  EQU   *-MQSD_SUBUSERDATA
ORG MQSD_SUBUSERDATA
MQSD_SUBUSERDATA_AREA    DS   CL(MQSD_SUBUSERDATA_LENGTH)
*
MQSD_SUBCORRELID         DS   CL24 Correlation Id related to this subscription
MQSD_PUBPRIORITY         DS   F   Priority set in publications
MQSD_PUBACCOUNTINGTOKEN  DS   CL32 Accounting Token set in publications
MQSD_PUBAPPLIDENTITYDATA DS   CL32 Appl Identity Data set in publications
*
MQSD_SELECTIONSTRING     DS   F   Message Selector
MQSD_SELECTIONSTRING_VSPTR DS   F   Address of variable length string
MQSD_SELECTIONSTRING_VSOFFSET DS   F   Offset of variable length string
MQSD_SELECTIONSTRING_VSBUFFSIZE DS   F   size of buffer
MQSD_SELECTIONSTRING_VSLNGTH DS   F   Length of variable length string
MQSD_SELECTIONSTRING_VSCCSID DS   F   CCSID of variable length string
MQSD_SELECTIONSTRING_LENGTH EQU   *-MQSD_SELECTIONSTRING
ORG MQSD_SELECTIONSTRING
MQSD_SELECTIONSTRING_AREA DS   CL(MQSD_SELECTIONSTRING_LENGTH)
*
MQSD-SUBLEVEL            DS   F   Subscription level
*
MQSD_RESOBJECTSTRING     DS   F   Resolved Long object name
MQSD_RESOBJECTSTRING_VSPTR DS   F   Address of variable length string
MQSD_RESOBJECTSTRING_VSOFFSET DS   F   Offset of variable length string
MQSD_RESOBJECTSTRING_VSBUFFSIZE DS   F   size of buffer
MQSD_RESOBJECTSTRING_VSLNGTH DS   F   Length of variable length string
MQSD_RESOBJECTSTRING_VSCCSID DS   F   CCSID of variable length string
MQSD_RESOBJECTSTRING_LENGTH EQU   *-MQSD_RESOBJECTSTRING
ORG MQSD_RESOBJECTSTRING
MQSD_RESOBJECTSTRING_AREA DS   CL(MQSD_RESOBJECTSTRING_LENGTH)
*
MQSD_LENGTH              EQU   *-MQSD
ORG MQSD
MQSD_AREA                 DS   CL(MQSD_LENGTH)

```

MQSMPO - Set message property options

The following table summarizes the fields in the structure.

Table 537. Fields in MQSMPO

Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options
<i>ValueEncoding</i>	Property value encoding	ValueEncoding
<i>ValueCCSID</i>	Property value character set	ValueCCSID

Overview for MQSMPO

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The **MQSMPO** structure allows applications to specify options that control how properties of messages are set. The structure is an input parameter on the **MQSETMP** call.

Character set and encoding: Data in **MQSMPO** must be in the character set of the application and encoding of the application (**MQENC_NATIVE**).

Fields for MQSMPO

The MQSMPO structure contains the following fields; the fields are described in **alphabetical order**:

Options (MQLONG)

Location options: The following options relate to the relative location of the property compared to the property cursor:

MQSMPO_SET_FIRST

Sets the value of the first property that matches the specified name, or if it does not exist, adds a new property after all other properties with a matching hierarchy.

MQSMPO_SET_PROP_UNDER_CURSOR

Sets the value of the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired using either the MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused on an MQGET call, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established or if the property pointer to by the property cursor has been deleted, the call fails with completion code MQCC_FAILED and reason code MQRC_PROPERTY_NOT_AVAILABLE.

MQSMPO_SET_PROP_BEFORE_CURSOR

Sets a new property before the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired using either the MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused on an MQGET call, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established or if the property pointer to by the property cursor has been deleted, the call fails with completion code MQCC_FAILED and reason code MQRC_PROPERTY_NOT_AVAILABLE.

MQSMPO_SET_PROP_AFTER_CURSOR

Sets a new property after the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired using either the MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused on an MQGET call, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established or if the property pointer to by the property cursor has been deleted, the call fails with completion code MQCC_FAILED and reason code MQRC_PROPERTY_NOT_AVAILABLE.

MQSMPO_APPEND_PROPERTY

Causes a new property to be added after all other properties with a matching hierarchy. If at least one property exists which matches the specified name, then a new property is added at the end after the end of that list of properties.

This option allows a list of properties with the same name to be created.

If you need none of the options described, use the following option:

MQSMPO_NONE

No options specified.

This is always an input field. The initial value of this field is MQSMPO_SET_FIRST.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQSMPO_STRUC_ID

Identifier for set message property options structure.

For the C programming language, the constant **MQSMPO_STRUC_ID_ARRAY** is also defined; this has the same value as **MQSMPO_STRUC_ID**, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is **MQSMPO_STRUC_ID**.

ValueCCSID (MQLONG)

The character set of the property value to be set if the value is a character string.

This is always an input field. The initial value of this field is **MQCCSI_APPL**.

ValueEncoding (MQLONG)

The encoding of the property value to be set if the value is numeric.

This is always an input field. The initial value of this field is **MQENC_NATIVE**.

Version (MQLONG)

This is the structure version number; the value must be:

MQSMPO_VERSION_1

Version-1 set message property options structure.

The following constant specifies the version number of the current version:

MQSMPO_CURRENT_VERSION

Current version of set message property options structure.

This is always an input field. The initial value of this field is **MQSMPO_VERSION_1**.

Initial values and language declarations for MQSMPO

Table 538. Initial values of fields in MQSMPO		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQSMPO_STRUC_ID	'SMPO'
<i>Version</i>	MQSMPO_VERSION_1	1
<i>Options</i>	MQSMPO_NONE	0
<i>ValueEncoding</i>	MQENC_NATIVE	Depends on environment
<i>ValueCCSID</i>	MQCCSI_APPL	-3
Notes: 1. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 2. In the C programming language, the macro variable MQSMPO_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQSMPO MySMPO = {MQSMPO_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQSMPO MQSMPO;
struct tagMQSMPO {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     Options;          /* Options that control the action of MQSETMP */
    MQLONG     ValueEncoding;     /* Encoding of Value */
    MQLONG     ValueCCSID;       /* Character set identifier of Value */
};
```

COBOL declaration

```
** MQSMPO structure
10 MQSMPO.
**   Structure identifier
15 MQSMPO-STRUCID      PIC X(4).
**   Structure version number
15 MQSMPO-VERSION     PIC S9(9) BINARY.
**   Options that control the action of MQSETMP
15 MQSMPO-OPTIONS     PIC S9(9) BINARY.
**   Encoding of VALUE
15 MQSMPO-VALUEENCODING PIC S9(9) BINARY.
**   Character set identifier of VALUE
15 MQSMPO-VALUECCSID  PIC S9(9) BINARY.
```

PL/I declaration

```
dcl
1 MQSMPO based,
3 StrucId      char(4),           /* Structure identifier */
3 Version      fixed bin(31),    /* Structure version number */
3 Options      fixed bin(31),    /* Options that control the action of MQSETMP */
3 ValueEncoding fixed bin(31),   /* Encoding of Value */
3 ValueCCSID   fixed bin(31),    /* Character set identifier of Value */
```

High Level Assembler declaration

```
MQSMPO          DSECT
MQSMPO_STRUCID  DS    CL4    Structure identifier
MQSMPO_VERSION  DS    F      Structure version number
MQSMPO_OPTIONS  DS    F      Options that control the action of
*                                     MQSETMP
MQSMPO_VALUEENCODING DS    F    Encoding of VALUE
MQSMPO_VALUECCSID DS    F      Character set identifier of VALUE
MQSMPO_LENGTH   EQU    *-MQSMPO
MQSMPO_AREA     DS    CL(MQSMPO_LENGTH)
```

MQSRO - Subscription request options

This section describes subscription request options, what fields it contains, and initial values of those fields.

Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>Options</i>	Options	Options
<i>NumPubs</i>	Number of publications	NumPubs

Overview for MQSRO

Availability: AIX, HP-UX, IBM i, Solaris, Linux, Windows, z/OS plus IBM MQ MQI clients connected to these systems.

Purpose: The MQSRO structure allows the application to specify options that control how a subscription request is made. The structure is an input/output parameter on the MQSUBRQ call.

Version: The current version of MQSRO is MQSRO_VERSION_1.

Character set and encoding: Data in MQSRO must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE. However, if the application is running as an MQ MQI client, the structure must be in the character set and encoding of the client.

Fields for MQSRO

The MQSRO structure contains the following fields; the fields are described in alphabetical order:

NumPubs (MQLONG)

This is an output field, returned to the application to indicate the number of publications sent to the subscription queue as a result of this call. Although this number of publications have been sent as a result of this call, there is no guarantee that this many messages will be available for the application to get, especially if they are non-persistent messages.

There might be more than one publication if the topic subscribed to contained a wildcard. If no wildcards were present in the topic string when the subscription represented by *Hsub* was created, then at most one publication is sent as a result of this call.

Options (MQLONG)

One of the following options must be specified. Only one option can be specified.

MQSRO_FAIL_IF QUIESCING

The MQSUBRQ call fails if the queue manager is in the quiescing state. On z/OS, for a CICS or IMS application, this option also forces the MQSUBRQ call to fail if the connection is in a quiescing state.

Default option: If the option described previously is not required, the following option must be used:

MQSRO_NONE

Use this value to indicate that no other options have been specified; all options assume their default values.

MQSRO_NONE helps program documentation. Although it is not intended that this option be used with any other, because its value is zero, this use cannot be detected.

StrucId (MQCHAR4)

This is the structure identifier; the value must be:

MQSRO_STRUC_ID

Identifier for Subscription Request Options structure.

For the C programming language, the constant MQSRO_STRUC_ID_ARRAY is also defined; this has the same value as MQSRO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQSRO_STRUC_ID.

Version (MQLONG)

This is the structure version number; the value must be:

MQSRO_VERSION_1

Version-1 Subscription Request Options structure.

The following constant specifies the version number of the current version:

MQSRO_CURRENT_VERSION

Current version of Subscription Request Options structure.

This is always an input field. The initial value of this field is MQSRO_VERSION_1.

Initial values and language declarations for MQSRO

Field name	Name of constant	Value of constant
<i>StrucId</i>	MQSRO_STRUC_ID	'SR0~'
<i>Version</i>	MQSRO_VERSION_1	1
<i>Options</i>	MQSRO_NONE	0
<i>NumPubs</i>	None	0
Notes: 1. The symbol ~ represents a single blank character. 2. In the C programming language, the macro variable MQSRO_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure: <pre>MQSRO MySRO = {MQSRO_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQSRO MQSRO;  
struct tagMQSRO {  
    MQCHAR4    StrucId;          /* Structure identifier */
```

```

MQLONG    Version;           /* Structure version number */
MQLONG    Options;          /* Options that control the action of MQSUBRQ */
MQLONG    NumPubs;          /* Number of publications sent */
/* Ver:1 */
};

```

COBOL declaration

```

** MQSRO structure
10 MQSRO.
** Structure identifier
15 MQSRO-STRUCID          PIC X(4).
** Structure version number
15 MQSRO-VERSION          PIC S9(9) BINARY.
** Options that control the action of MQSUBRQ
15 MQSRO-OPTIONS          PIC S9(9) BINARY.
** Number of publications sent
15 MQSRO-NUMPUBS          PIC S9(9) BINARY.

```

PL/I declaration

```

dcl
1 MQSRO based,
3 StrucId          char(4),          /* Structure identifier */
3 Version          fixed bin(31),    /* Structure version number */
3 Options          fixed bin(31),    /* Options that control the action of MQSUBRQ */
3 NumPubs          fixed bin(31);    /* Number of publications sent */

```

High Level Assembler declaration

```

MQSRO          DSECT
MQSRO_STRUCID  DS    CL4    Structure identifier
MQSRO_VERSION  DS    F      Structure version number
MQSRO_OPTIONS  DS    F      Options that control the action of MQSUBRQ
MQSRO_NUMPUBS  DS    F      Number of publications sent
*
MQSRO_LENGTH   EQU    *-MQSRO
ORG    MQSRO
MQSRO_AREA     DS      CL(MQSRO_LENGTH)

```

MQSTS - Status reporting structure

The following table summarizes the fields in the structure.

Table 539. Fields in MQSTS		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>CompCode</i>	Completion code of first error	CompCode
<i>Reason</i>	Reason code of first error	Reason
<i>PutSuccessCount</i>	Number of successful asynchronous put calls	SuccessCount
<i>PutWarningCount</i>	Number of asynchronous put calls which had warnings	WarningCount
<i>PutFailureCount</i>	Number of failed asynchronous put calls	FailureCount
<i>ObjectType</i>	Type of failing object	ObjectType
<i>ObjectName</i>	Name of failing object	ObjectName
<i>ObjectQMgrName</i>	Name of queue manager owning the failing object	ObjectQMgrName

Table 539. Fields in MQSTS (continued)		
Field	Description	Topic
<i>ResolvedObjectName</i>	Resolved name of destination queue	ResolvedObjectName
<i>ResolvedQMgrName</i>	Resolved name of destination queue manager	ResolvedQMgrName
Note: The remaining fields are ignored if Version is less than MQSTS_VERSION_2.		
<i>ObjectString</i>	Long object name of failing object	ObjectString
<i>SubName</i>	Subscription name of failing subscription	SubName
<i>OpenOptions</i>	Open options associated with the failure	OpenOptions
<i>SubOptions</i>	Subscription options associated with the failure	SubOptions

Overview for MQSTS

Purpose: The MQSTS structure is an output parameter from the MQSTAT command.

Character set and encoding: Character data in MQSTS is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in MQSTS is in the native machine encoding; this is given by *Encoding*.

Usage: The MQSTAT command is used to retrieve status information. This information is returned in an MQSTS structure. For information about MQSTAT, see [“MQSTAT - Retrieve status information”](#) on page 754.

Fields for MQSTS

The MQSTS structure contains the following fields; the fields are described in **alphabetical order**:

CompCode (MQLONG)

The completion code of the operation being reported on.

The interpretation of *CompCode* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

This is the completion code resulting from a previous asynchronous put operation on the object specified in *ObjectName*.

MQSTAT_TYPE_RECONNECTION

If the connection is reconnecting or failed to reconnect this is the completion code that caused the connection to begin reconnecting.

If the connection is currently connected the value is MQCC_OK.

MQSTAT_TYPE_RECONNECTION_ERROR

If the connection failed to reconnect this is the completion code that caused the reconnection to fail.

If the connection is currently connected, or reconnecting, the value is MQCC_OK.

CompCode is always an output field. Its initial value is MQCC_OK.

ObjectName (MQCHAR48)

The name of the object being reported on.

The interpretation of *ObjectName* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

This is the name of the queue or topic used in the put operation, the failure of which is reported in the *CompCode* and *Reason* fields in the MQSTS structure.

MQSTAT_TYPE_RECONNECTION

If the connection is reconnecting, this is the name of the queue manager associated with the connection.

MQSTAT_TYPE_RECONNECTION_ERROR

If the connection failed to reconnect, this is the name of the object which caused reconnection to fail. The reason for the failure is reported in the *CompCode* and *Reason* fields in the MQSTS structure.

ObjectName is an output field. Its initial value is the null string in C, and 48 blank characters in other programming languages.

ObjectQMgrName (MQCHAR48)

The name of the queue manager being reported on.

The interpretation of *ObjectQMgrName* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

This is the name of the queue manager on which the *ObjectName* object is defined. A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected (the local queue manager).

MQSTAT_TYPE_RECONNECTION

Blank.

MQSTAT_TYPE_RECONNECTION_ERROR

If the connection failed to reconnect, this is the name of the object which caused reconnection to fail. The reason for the failure is reported in the *CompCode* and *Reason* fields in the MQSTS structure.

ObjectQMgrName is an output field. Its value is the null string in C, and 48 blank characters in other programming languages.

ObjectString (MQCHARV)

Long object name of failing object being reported on. Present only in Version 2 of MQSTS or higher.

The interpretation of *ObjectString* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

This is the long object name of the queue or topic used in the MQPUT operation, which failed.

MQSTAT_TYPE_RECONNECTION

Zero length string

MQSTAT_TYPE_RECONNECTION_ERROR

This is the long object name of the object that caused the reconnection to fail.

ObjectString is an output field. Its initial value is a zero length string.

ObjectType (MQLONG)

The type of the object named in *ObjectName* being reported on.

Possible values of *ObjectType* are listed in [“MQOT_* \(Object Types and Extended Object Types\)” on page 160](#).

ObjectType is an output field. Its initial value is MQOT_Q.

OpenOptions (MQLONG)

The OpenOptions used to open the object being reported upon. Present only in Version 2 of MQSTS or higher.

The value of OpenOptions depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

Zero.

MQSTAT_TYPE_RECONNECTION

Zero.

MQSTAT_TYPE_RECONNECTION_ERROR

The OpenOptions used when the failure occurred. The reason for the failure is reported in the *CompCode* and *Reason* fields in the MQSTS structure.

OpenOptions is an output field. Its initial value is zero.

PutFailureCount (MQLONG)

The number of asynchronous put operations that failed.

The value of PutFailureCount depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

The number of asynchronous put operations to the object named in the MQSTS structure that completed with MQCC_FAILED.

MQSTAT_TYPE_RECONNECTION

Zero.

MQSTAT_TYPE_RECONNECTION_ERROR

Zero.

PutFailureCount is an output field. Its initial value is zero.

PutSuccessCount (MQLONG)

The number of asynchronous put operations that succeeded.

The value of PutSuccessCount depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

The number of asynchronous put operations to the object named in the MQSTS structure that completed with MQCC_OK.

MQSTAT_TYPE_RECONNECTION

Zero.

MQSTAT_TYPE_RECONNECTION_ERROR

Zero.

PutSuccessCount is an output field. Its initial value is zero.

PutWarningCount (MQLONG)

The number of asynchronous put operations that ended with a warning.

The value of PutWarningCount depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

The number of asynchronous put operations to the object named in the MQSTS structure that completed with MQCC_WARNING.

MQSTAT_TYPE_RECONNECTION

Zero.

MQSTAT_TYPE_RECONNECTION_ERROR

Zero.

PutWarningCount is an output field. Its initial value is zero.

SubName (MQCHARV)

The name of the failing subscription. Present only in Version 2 of MQSTS or higher.

The interpretation of SubName depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

Zero length string.

MQSTAT_TYPE_RECONNECTION

Zero length string.

MQSTAT_TYPE_RECONNECTION_ERROR

The name of the subscription that caused reconnection to fail. If no subscription name is available, or the failure is not related to a subscription, this is a zero-length string.

SubName is an output field. Its initial value is a zero length string.

SubOptions (MQLONG)

The SubOptions used to open the failing subscription. Present only in Version 2 of MQSTS or higher.

The interpretation of SubOptions depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

Zero.

MQSTAT_TYPE_RECONNECTION

Zero.

MQSTAT_TYPE_RECONNECTION_ERROR

The SubOptions used when the failure occurred. If the failure is not related to subscribing to a topic, the value returned is zero.

SubOptions is an output field. Its initial value is zero.

Reason (MQLONG)

The reason code of the operation being reported on.

The interpretation of Reason depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

This is the reason code resulting from a previous asynchronous put operation on the object specified in ObjectName.

MQSTAT_TYPE_RECONNECTION

If the connection is reconnecting or failed to reconnect this is the reason code that caused the reconnection to begin reconnecting.

If the connection is currently connected the value is MQRC_NONE.

MQSTAT_TYPE_RECONNECTION_ERROR

If the connection failed to reconnect this is the reason code that caused the reconnection to fail.

If the connection is currently connected, or reconnecting, the value is MQRC_NONE.

Reason is an output field. Its initial value is MQRC_NONE.

ResolvedObjectName (MQCHAR48)

The name of the object named in *ObjectName* after the local queue manager resolves the name.

The interpretation of *ResolvedObjectName* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

ResolvedObjectName is the name of the object named in *ObjectName* after the local queue manager resolves the name. The name returned is the name of an object that exists on the queue manager identified by *ResolvedQMGrName*.

MQSTAT_TYPE_RECONNECTION

Blank.

MQSTAT_TYPE_RECONNECTION_ERROR

Blank.

ResolvedObjectName is an output field. Its initial value is the null string in C, and 48 blank characters in other programming languages.

ResolvedQMGrName (MQCHAR48)

The name of the destination queue manager after the local queue manager resolves the name.

The interpretation of *ResolvedQMGrName* depends on the value of the MQSTAT Type parameter.

MQSTAT_TYPE_ASYNC_ERROR

ResolvedQMGrName is the name of the destination queue manager after the local queue manager resolves the name. The name returned is the name of the queue manager that owns the object identified by *ResolvedObjectName*. *ResolvedQMGrName* might be the name of the local queue manager.

MQSTAT_TYPE_RECONNECTION

Blank.

MQSTAT_TYPE_RECONNECTION_ERROR

Blank.

ResolvedQMGrName is always an output field. Its initial value is the null string in C, and 48 blank characters in other programming languages.

StrucId (MQCHAR4)

The identifier for the status reporting structure, MQSTS.

StrucId is the structure identifier. The value must be:

MQSTS_STRUC_ID

Identifier for status reporting structure.

For the C programming language, the constant MQSTS_STRUC_ID_ARRAY is also defined; this has the same value as MQSTS_STRUC_ID, but is an array of characters instead of a string.

StrucId is always an input field. Its initial value is MQSTS_STRUC_ID.

Version (MQLONG)

The structure version number.

The value must be either:

MQSTS_VERSION_1

Version 1 status reporting structure.

MQSTS_VERSION_2

Version 2 status reporting structure.

The following constant specifies the version number of the current version:

MQSTS_CURRENT_VERSION

Current version of status reporting structure. The current version is MQSTS_VERSION_2.

Version is always an input field. Its initial value is MQSTS_VERSION_1.

Initial values and language declarations for MQSTS

Table 540. Initial values of fields in MQSTS		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQSTS_STRUC_ID	'STAT↵'
<i>Version</i>	MQSTS_VERSION_1	1
<i>CompCode</i>	MQCC_OK	0
<i>Reason</i>	MQRC_NONE	0
<i>PutSuccessCount</i>	None	0
<i>PutWarningCount</i>	None	0
<i>PutFailureCount</i>	None	0
<i>ObjectType</i>	MQOT_Q	1
<i>ObjectName</i>	None	Null string or blanks
<i>ObjectQMgrName</i>	None	Null string or blanks
<i>ResolvedObjectName</i>	None	Null string or blanks
<i>ResolvedQMgrName</i>	None	Null string or blanks
<i>ObjectString</i>	MQCHARV_DEFAULT	{NULL,0,0,0,-3}
<i>SubName</i>	MQCHARV_DEFAULT	{NULL,0,0,0,-3}
<i>OpenOptions</i>	None	0
<i>SubOptions</i>	None	0
Notes: <ol style="list-style-type: none">1. The symbol ↵ represents a single blank character.2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages.3. In the C programming language, the macro variable MQSTS_DEFAULT contains the values that are listed in the table. It can be used in the following way to provide initial values for the fields in the structure:<pre>MQSTS MySTS = {MQSTS_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQSTS MQSTS;
struct tagMQSTS {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;           /* Structure version number */
    MQLONG     CompCode;          /* Completion Code of first error */
    MQLONG     Reason;            /* Reason Code of first error */
    MQLONG     PutSuccessCount;    /* Number of Async calls succeeded */
```

```

MQLONG    PutWarningCount;    /* Number of Async calls had warnings */
MQLONG    PutFailureCount;    /* Number of Async calls had failures */
MQLONG    ObjectType;         /* Failing object type */
MQCHAR48  ObjectName;         /* Failing object name */
MQCHAR48  ObjectQMgrName;     /* Failing object queue manager name */
MQCHAR48  ResolvedObjectName; /* Resolved name of destination queue */
MQCHAR48  ResolvedQMgrName;   /* Resolved name of destination qmgr */
/* Ver:1 */
MQCHARV   ObjectString;       /* Failing object long name */
MQCHARV   SubName;            /* Failing subscription name */
MQLONG    OpenOptions;        /* Failing open options */
MQLONG    SubOptions;         /* Failing subscription options */
/* Ver:2 */
};

```

COBOL declaration

```

** MQSTS structure
10 MQSTS.
** Structure identifier
15 MQSTS-STRUCID PIC X(4).
** Structure version number
15 MQSTS-VERSION PIC S9(9) BINARY.
** Completion Code of first error
15 MQSTS-COMPCODE PIC S9(9) BINARY.
** Reason Code of first error
15 MQSTS-REASON PIC S9(9) BINARY.
** Number of Async put calls succeeded
15 MQSTS-PUTSUCCESSCOUNT PIC S9(9) BINARY.
** Number of Async put calls had warnings
15 MQSTS-PUTWARNINGCOUNT PIC S9(9) BINARY.
** Number of Async put calls had failures
15 MQSTS-PUTFAILURECOUNT PIC S9(9) BINARY.
** Failing object type
15 MQSTS-OBJECTTYPE PIC S9(9) BINARY.
** Failing object name
15 MQSTS-OBJECTNAME PIC X(48).
** Failing object queue manager
15 MQSTS-OBJECTQMGRNAME PIC X(48).
** Resolved name of destination queue
15 MQSTS-RESOLVEDOBJECTNAME PIC X(48).
** Resolved name of destination qmgr
15 MQSTS-RESOLVEDQMGRNAME PIC X(48).
** Ver:1 **
** Failing object long name
15 MQSTS-OBJECTSTRING.
** Address of variable length string
20 MQSTS-OBJECTSTRING-VSPTR POINTER.
** Offset of variable length string
20 MQSTS-OBJECTSTRING-VSOFFSET PIC S9(9) BINARY.
** Size of buffer
20 MQSTS-OBJECTSTRING-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQSTS-OBJECTSTRING-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSTS-OBJECTSTRING-VSCCSID PIC S9(9) BINARY.
** Failing subscription name
15 MQSTS-SUBNAME.
** Address of variable length string
20 MQSTS-SUBNAME-VSPTR POINTER.
** Offset of variable length string
20 MQSTS-SUBNAME-VSOFFSET PIC S9(9) BINARY.
** Size of buffer
20 MQSTS-SUBNAME-VSBUFSIZE PIC S9(9) BINARY.
** Length of variable length string
20 MQSTS-SUBNAME-VSLENGTH PIC S9(9) BINARY.
** CCSID of variable length string
20 MQSTS-SUBNAME-VSCCSID PIC S9(9) BINARY.
** Failing open options
15 MQSTS-OPENOPTIONS PIC S9(9) BINARY.
** Failing subscription options
15 MQSTS-SUBOPTIONS PIC S9(9) BINARY.
** Ver:2 **

```

PL/I declaration

```

dcl
1 MQSTS based,

```

```

3 StrucId          char(4),          /* Structure identifier */
3 Version          fixed bin(31),    /* Structure version number */
3 CompCode         fixed bin(31),    /* Completion code */
3 Reason           fixed bin(31),    /* Reason code */
3 PutSuccessCount  fixed bin(31),    /* Put success count */
3 PutWarningCount  fixed bin(31),    /* Put warning count */
3 PutFailureCount  fixed bin(31),    /* Put failure count */
3 ObjectType       fixed bin(31),    /* Object type */
3 ObjectName       char(48),         /* Object name */
3 ObjectQmgrName   char(48),         /* Object queue manager */
3 ResolvedObjectName char(48),      /* Resolved Object name */
3 ResolvedQmgrName char(48);        /* Resolved Object queue manager */
/* Ver:1 */
3 ObjectString,    /* Failing object long name */
5 VSPtr pointer,   /* Address of variable length string */
5 VSOffset fixed bin(31), /* Offset of variable length string */
5 VBufSize fixed bin(31), /* Size of buffer */
5 VSLength fixed bin(31), /* Length of variable length string */
5 VSCCSID fixed bin(31); /* CCSID of variable length string */
3 SubName,        /* Failing subscription name */
5 VSPtr pointer,   /* Address of variable length string */
5 VSOffset fixed bin(31), /* Offset of variable length string */
5 VBufSize fixed bin(31), /* Size of buffer */
5 VSLength fixed bin(31), /* Length of variable length string */
5 VSCCSID fixed bin(31); /* CCSID of variable length string */
3 OpenOptions fixed bin(31), /* Failing open options */
3 SubOptions fixed bin(31);  /* Failing subscription options */
/* Ver:2 */

```

High Level Assembler declaration

MQSTS	DSECT	
MQSTS_STRUCID	DS	CL4 Structure identifier
MQSTS_VERSION	DS	F Structure version number
MQSTS_COMPCODE	DS	F Completion code
MQSTS_REASON	DS	F Reason code
MQSTS_PUTSUCCESSCOUNT	DS	F Success count
MQSTS_PUTWARNINGCOUNT	DS	F Warning count
MQSTS_PUTFAILURECOUNT	DS	F Failure count
MQSTS_OBJTYPE	DS	F Object type
MQSTS_OBJNAME	DS	CL48 Object name
MQSTS_OBJQMGR	DS	CL48 Object queue manager
MQSTS_ROBJNAME	DS	CL48 Resolved object name
MQSTS_ROBJQMGR	DS	CL48 Resolved object queue manager
MQSTS_OBJECTSTRING	DS	0F Force fullword alignment
MQSTS_OBJECTSTRING_VSPTR	DS	A Address of variable length string
MQSTS_OBJECTSTRING_VSOFFSET	DS	F Offset of variable length string
MQSTS_OBJECTSTRING_VBUFSIZE	DS	F Size of buffer
MQSTS_OBJECTSTRING_VSLength	DS	F Length of variable length string
MQSTS_OBJECTSTRING_VSCCSID	DS	F CCSID of variable length string
MQSTS_OBJECTSTRING_LENGTH	EQU	*-MQSTS_OBJECTSTRING
		ORG MQSTS_OBJECTSTRING
MQSTS_OBJECTSTRING_AREA	DS	CL(MQSTS_OBJECTSTRING_LENGTH)
*		
MQSTS_SUBNAME	DS	0F Force fullword alignment
MQSTS_SUBNAME_VSPTR	DS	A Address of variable length string
MQSTS_SUBNAME_VSOFFSET	DS	F Offset of variable length string
MQSTS_SUBNAME_VBUFSIZE	DS	F Size of buffer
MQSTS_SUBNAME_VSLength	DS	F Length of variable length string
MQSTS_SUBNAME_VSCCSID	DS	F CCSID of variable length string
MQSTS_SUBNAME_LENGTH	EQ	*-MQSTS_SUBNAME
		ORG MQSTS_SUBNAME
MQSTS_SUBNAME_AREA	DS	CL(MQSTS_SUBNAME_LENGTH)
*		
MQSTS_OPENOPTIONS	DS	F Failing open options
MQSTS_SUBOPTIONS	DS	F Failing subscription option
MQSTS_LENGTH	EQU	*-MQSTS
		ORG MQSTS
MQSTS_AREA	DS	CL(MQSTS_LENGTH)

MQTM - Trigger message

The following table summarizes the fields in the structure.

Table 541. Fields in MQTM		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>QName</i>	Name of triggered queue	QName
<i>ProcessName</i>	Name of process object	ProcessName
<i>TriggerData</i>	Trigger data	TriggerData
<i>ApplType</i>	Application type	ApplType
<i>ApplId</i>	Application identifier	ApplId
<i>EnvData</i>	Environment data	EnvData
<i>UserData</i>	User data	UserData

Overview for MQTM

Purpose: The MQTM structure describes the data in the trigger message that is sent by the queue manager to a trigger-monitor application when a trigger event occurs for a queue.

This structure is part of the IBM MQ Trigger Monitor Interface (TMI), which is one of the IBM MQ framework interfaces.

Format name: MQFMT_TRIGGER.

Character set and encoding: Character data in MQTM is in the character set of the queue manager that generates the MQTM. Numeric data in MQTM is in the machine encoding of the queue manager that generates the MQTM.

The character set and encoding of the MQTM are given by the *CodedCharSetId* and *Encoding* fields in:

- The MQMD (if the MQTM structure is at the start of the message data), or
- The header structure that precedes the MQTM structure (all other cases).

Usage: A trigger-monitor application might need to pass some or all of the information in the trigger message to the application that the trigger-monitor application starts. Information that might be needed by the started application includes *QName*, *TriggerData*, and *UserData*. The trigger-monitor application can pass the MQTM structure directly to the started application, or pass an MQTMC2 structure instead, depending on what is permitted by the environment and convenient for the started application. For information about MQTMC2, see “MQTMC2 - Trigger message 2 (character format)” on page 579.

- On z/OS, for an MQAT_CICS application that is started using the CKTI transaction, the entire trigger message structure MQTM is made available to the started transaction; the information can be retrieved by using the EXEC CICS RETRIEVE command.
- On IBM i, the trigger-monitor application provided with IBM MQ passes an MQTMC2 structure to the started application.

For information about using triggers, see [Starting IBM MQ applications using triggers](#).

MQMD for a trigger message: The fields in the MQMD of a trigger message generated by the queue manager are set as follows:

Field in MQMD	Value used
<i>StrucId</i>	MQMD_STRUC_ID
<i>Version</i>	MQMD_VERSION_1
<i>Report</i>	MQRO_NONE
<i>MsgType</i>	MQMT_DATAGRAM

Field in MQMD	Value used
<i>Expiry</i>	MQEI_UNLIMITED
<i>Feedback</i>	MQFB_NONE
<i>Encoding</i>	MQENC_NATIVE
<i>CodedCharSetId</i>	Queue manager's <i>CodedCharSetId</i> attribute
<i>Format</i>	MQFMT_TRIGGER
<i>Priority</i>	Initiation queue's <i>DefPriority</i> attribute
<i>Persistence</i>	MQPER_NOT_PERSISTENT
<i>MsgId</i>	A unique value
<i>CorrelId</i>	MQCI_NONE
<i>BackoutCount</i>	0
<i>ReplyToQ</i>	Blanks
<i>ReplyToQMgr</i>	Name of queue manager
<i>UserIdentifier</i>	Blanks
<i>AccountingToken</i>	MQACT_NONE
<i>ApplIdentityData</i>	Blanks
<i>PutApplType</i>	MQAT_QMGR, or as appropriate for the message channel agent
<i>PutApplName</i>	First 28 bytes of the queue-manager name
<i>PutDate</i>	Date when trigger message is sent
<i>PutTime</i>	Time when trigger message is sent
<i>ApplOriginData</i>	Blanks

An application that generates a trigger message is recommended to set similar values, except for the following:

- The *Priority* field can be set to MQPRI_PRIORITY_AS_Q_DEF (the queue manager will change this to the default priority for the initiation queue when the message is put).
- The *ReplyToQMgr* field can be set to blanks (the queue manager will change this to the name of the local queue manager when the message it put).
- Set the context fields as appropriate for the application.

Fields for MQTM

The MQTM structure contains the following fields; the fields are described in **alphabetical order**:

ApplId (MQCHAR256)

This is a character string that identifies the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ApplId* attribute of the process object identified by the *ProcessName* field; see [“Attributes for process definitions” on page 834](#) for details of this attribute. The content of this data is of no significance to the queue manager.

The meaning of *ApplId* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ requires *ApplId* to be the name of an executable program. The following notes apply to the environments indicated:

- On z/OS, *ApplId* is:
 - A CICS transaction identifier, for applications started using the CICS trigger-monitor transaction CKTI

- An IMS transaction identifier, for applications started using the IMS trigger monitor CSQQTRMN
- On Windows systems, the program name can be prefixed with a drive and directory path.
- On IBM i, the program name can be prefixed with a library name and / character.
- On UNIX systems, the program name can be prefixed with a directory path.

The length of this field is given by MQ_PROCESS_APPL_ID_LENGTH. The initial value of this field is the null string in C, and 256 blank characters in other programming languages.

ApplType (MQLONG)

This identifies the nature of the program to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ApplType* attribute of the process object identified by the *ProcessName* field; see [“Attributes for process definitions” on page 834](#) for details of this attribute. The content of this data is of no significance to the queue manager.

ApplType can have one of the following standard values. User-defined types can also be used, but should be restricted to values in the range MQAT_USER_FIRST through MQAT_USER_LAST:

MQAT_AIX

AIX application (same value as MQAT_UNIX).

MQAT_BATCH

Batch application

MQAT_BROKER

Broker application

MQAT_CICS

CICS transaction.

MQAT_CICS_BRIDGE

CICS bridge application.

MQAT_CICS_VSE

CICS/VSE transaction.

MQAT_DOS

IBM MQ MQI client application on PC DOS.

MQAT_IMS

IMS application.

MQAT_IMS_BRIDGE

IMS bridge application.

MQAT_JAVA

Java application.

MQAT_MVS

MVS or TSO application (same value as MQAT_ZOS).

MQAT_NOTES_AGENT

Lotus Notes Agent application.

MQAT_NSK

HP Integrity NonStop Server application.

MQAT_OS390

OS/390 application (same value as MQAT_ZOS).

MQAT_OS400

IBM i application.

MQAT_RRS_BATCH

RRS batch application.

MQAT_UNIX

UNIX application.

MQAT_UNKNOWN

Application of unknown type.

MQAT_USER

User-defined application type.

MQAT_VOS

Stratus VOS application.

MQAT_WINDOWS

16-bit Windows application.

MQAT_WINDOWS_NT

32-bit Windows application.

MQAT_WLM

z/OS workload manager application.

MQAT_XCF

XCF.

MQAT_ZOS

z/OS application.

MQAT_USER_FIRST

Lowest value for user-defined application type.

MQAT_USER_LAST

Highest value for user-defined application type.

The initial value of this field is 0.

EnvData (MQCHAR128)

This is a character string that contains environment-related information pertaining to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *EnvData* attribute of the process object identified by the *ProcessName* field; see “Attributes for process definitions” on [page 834](#) for details of this attribute. The content of this data is of no significance to the queue manager.

On z/OS, for a CICS application started using the CKTI transaction, or an IMS application to be started using the CSQQTRMN transaction, this information is not used.

The length of this field is given by MQ_PROCESS_ENV_DATA_LENGTH. The initial value of this field is the null string in C, and 128 blank characters in other programming languages.

ProcessName (MQCHAR48)

This is the name of the queue-manager process object specified for the triggered queue, and can be used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ProcessName* attribute of the queue identified by the *QName* field; see “Attributes for queues” on [page 801](#) for details of this attribute.

Names that are shorter than the defined length of the field are always padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by MQ_PROCESS_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

QName (MQCHAR48)

This is the name of the queue for which a trigger event occurred, and is used by the application started by the trigger-monitor application. The queue manager initializes this field with the value of the *QName* attribute of the triggered queue; see “Attributes for queues” on [page 801](#) for details of this attribute.

Names that are shorter than the defined length of the field are padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQTM_STRUC_ID

Identifier for trigger message structure.

For the C programming language, the constant MQTM_STRUC_ID_ARRAY is also defined; this has the same value as MQTM_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQTM_STRUC_ID.

TriggerData (MQCHAR64)

This is free-format data for use by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *TriggerData* attribute of the queue identified by the *QName* field; see “Attributes for queues” on page 801 for details of this attribute. The content of this data is of no significance to the queue manager.

On z/OS, for a CICS application started using the CKTI transaction, this information is not used.

The length of this field is given by MQ_TRIGGER_DATA_LENGTH. The initial value of this field is the null string in C, and 64 blank characters in other programming languages.

UserData (MQCHAR128)

This is a character string that contains user information relevant to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *UserData* attribute of the process object identified by the *ProcessName* field; see “Attributes for process definitions” on page 834 for details of this attribute. The content of this data is of no significance to the queue manager.

For Microsoft Windows, the character string must not contain double quotation marks if the process definition is going to be passed to **runmqtm**.

The length of this field is given by MQ_PROCESS_USER_DATA_LENGTH. The initial value of this field is the null string in C, and 128 blank characters in other programming languages.

Version (MQLONG)

This is the structure version number. The value must be:

MQTM_VERSION_1

Version number for trigger message structure.

The following constant specifies the version number of the current version:

MQTM_CURRENT_VERSION

Current version of trigger message structure.

The initial value of this field is MQTM_VERSION_1.

Initial values and language declarations for MQTM

Table 542. Initial values of fields in MQTM for MQTM		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQTM_STRUC_ID	'TM--'
<i>Version</i>	MQTM_VERSION_1	1

Table 542. Initial values of fields in MQTM for MQTM (continued)

Field name	Name of constant	Value of constant
<i>QName</i>	None	Null string or blanks
<i>ProcessName</i>	None	Null string or blanks
<i>TriggerData</i>	None	Null string or blanks
<i>ApplType</i>	None	0
<i>ApplId</i>	None	Null string or blanks
<i>EnvData</i>	None	Null string or blanks
<i>UserData</i>	None	Null string or blanks

Notes:

1. The symbol ~ represents a single blank character.
2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQTM_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure:

```
MQTM MyTM = {MQTM_DEFAULT};
```

C declaration

```
typedef struct tagMQTM MQTM;
struct tagMQTM {
    MQCHAR4    StrucId;        /* Structure identifier */
    MQLONG     Version;       /* Structure version number */
    MQCHAR48   QName;         /* Name of triggered queue */
    MQCHAR48   ProcessName;   /* Name of process object */
    MQCHAR64   TriggerData;   /* Trigger data */
    MQLONG     ApplType;      /* Application type */
    MQCHAR256  ApplId;        /* Application identifier */
    MQCHAR128  EnvData;       /* Environment data */
    MQCHAR128  UserData;      /* User data */
};
```

COBOL declaration

```
**  MQTM structure
10 MQTM.
**  Structure identifier
15 MQTM-STRUCID    PIC X(4).
**  Structure version number
15 MQTM-VERSION    PIC S9(9) BINARY.
**  Name of triggered queue
15 MQTM-QNAME      PIC X(48).
**  Name of process object
15 MQTM-PROCESSNAME PIC X(48).
**  Trigger data
15 MQTM-TRIGGERDATA PIC X(64).
**  Application type
15 MQTM-APPLTYPE   PIC S9(9) BINARY.
**  Application identifier
15 MQTM-APPLID     PIC X(256).
**  Environment data
15 MQTM-ENVDATA    PIC X(128).
**  User data
15 MQTM-USERDATA   PIC X(128).
```

PL/I declaration

```
dcl
  1 MQTM based,
    3 StrucId      char(4),          /* Structure identifier */
    3 Version      fixed bin(31),    /* Structure version number */
    3 QName        char(48),         /* Name of triggered queue */
    3 ProcessName  char(48),         /* Name of process object */
    3 TriggerData  char(64),         /* Trigger data */
    3 ApplType     fixed bin(31),    /* Application type */
    3 ApplId       char(256),        /* Application identifier */
    3 EnvData      char(128),        /* Environment data */
    3 UserData     char(128);        /* User data */
```

High Level Assembler declaration

```
MQTM          DSECT
MQTM_STRUCID   DS    CL4    Structure identifier
MQTM_VERSION   DS    F      Structure version number
MQTM_QNAME     DS    CL48   Name of triggered queue
MQTM_PROCESSNAME DS    CL48 Name of process object
MQTM_TRIGGERDATA DS    CL64 Trigger data
MQTM_APPLTYPE  DS    F      Application type
MQTM_APPLID    DS    CL256  Application identifier
MQTM_ENVDATA   DS    CL128  Environment data
MQTM_USERDATA  DS    CL128  User data
*
MQTM_LENGTH    EQU    *-MQTM
ORG    MQTM
MQTM_AREA      DS      CL(MQTM_LENGTH)
```

Visual Basic declaration

```
Type MQTM
  StrucId      As String*4    'Structure identifier'
  Version      As Long        'Structure version number'
  QName        As String*48   'Name of triggered queue'
  ProcessName  As String*48   'Name of process object'
  TriggerData  As String*64   'Trigger data'
  ApplType     As Long        'Application type'
  ApplId       As String*256  'Application identifier'
  EnvData      As String*128  'Environment data'
  UserData     As String*128  'User data'
End Type
```

MQTMC2 - Trigger message 2 (character format)

The following table summarizes the fields in the structure.

Table 543. Fields in MQTMC2		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>QName</i>	Name of triggered queue	QName
<i>ProcessName</i>	Name of process object	ProcessName
<i>TriggerData</i>	Trigger data	TriggerData
<i>ApplType</i>	Application type	ApplType
<i>ApplId</i>	Application identifier	ApplId
<i>EnvData</i>	Environment data	EnvData
<i>UserData</i>	User data	UserData

Table 543. Fields in MQTMC2 (continued)		
Field	Description	Topic
<i>QMgrName</i>	Queue manager name	<u>QMgrName</u>

Overview for MQTMC2

Purpose: When a trigger-monitor application retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor might need to pass some or all of the information in the trigger message to the application that the trigger monitor starts.

Information that the started application might need includes *QName*, *TriggerData*, and *UserData*. The trigger monitor application can pass the MQTM structure directly to the started application, or pass an MQTMC2 structure instead, depending on what is permitted by the environment and convenient for the started application.

This structure is part of the IBM MQ Trigger Monitor Interface (TMI), which is one of the IBM MQ framework interfaces.

Character set and encoding: Character data in MQTMC2 is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute.

Usage: The MQTMC2 structure is very similar to the format of the MQTM structure. The difference is that the non-character fields in MQTM are changed in MQTMC2 to character fields of the same length, and the queue manager name is added at the end of the structure.

- On z/OS, for an MQAT_IMS application that is started using the CSQQTRMN application, an MQTMC2 structure is made available to the started application.
- On IBM i, the trigger monitor application provided with IBM MQ passes an MQTMC2 structure to the started application.

Fields for MQTMC2

The MQTMC2 structure contains the following fields; the fields are described in **alphabetical order**:

ApplId (MQCHAR256)

Application identifier.

See the *ApplId* field in the MQTM structure.

ApplType (MQCHAR4)

Application type.

This field always contains blanks, whatever the value in the *ApplType* field in the MQTM structure of the original trigger message.

EnvData (MQCHAR128)

Environment data.

See the *EnvData* field in the MQTM structure.

ProcessName (MQCHAR48)

Name of process object.

See the *ProcessName* field in the MQTM structure.

QMgrName (MQCHAR48)

Queue manager name.

This is the name of the queue manager at which the trigger event occurred.

QName (MQCHAR48)

Name of triggered queue.

See the *QName* field in the MQTM structure.

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQTMC_STRUC_ID

Identifier for trigger message (character format) structure.

For the C programming language, the constant MQTMC_STRUC_ID_ARRAY is also defined; this has the same value as MQTMC_STRUC_ID, but is an array of characters instead of a string.

TriggerData (MQCHAR64)

Trigger data.

See the *TriggerData* field in the MQTM structure.

UserData (MQCHAR128)

User data.

See the *UserData* field in the MQTM structure.

Version (MQCHAR4)

Structure version number.

The value must be:

MQTMC_VERSION_2

Version 2 trigger message (character format) structure.

For the C programming language, the constant MQTMC_VERSION_2_ARRAY is also defined; this has the same value as MQTMC_VERSION_2, but is an array of characters instead of a string.

The following constant specifies the version number of the current version:

MQTMC_CURRENT_VERSION

Current version of trigger message (character format) structure.

Initial values and language declarations for MQTMC2

Table 544. Initial values of fields in MQTMC2 for MQTMC2		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQTMC_STRUC_ID	'TMC↵'
<i>Version</i>	MQTMC_VERSION_2	'↵↵↵2'
<i>QName</i>	None	Null string or blanks
<i>ProcessName</i>	None	Null string or blanks
<i>TriggerData</i>	None	Null string or blanks
<i>ApplType</i>	None	Blanks
<i>ApplId</i>	None	Null string or blanks
<i>EnvData</i>	None	Null string or blanks
<i>UserData</i>	None	Null string or blanks

Table 544. Initial values of fields in MQTMC2 for MQTMC2 (continued)

Field name	Name of constant	Value of constant
QMgrName	None	Null string or blanks
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQTMC2_DEFAULT contains the values listed above. Use it in the following way to provide initial values for the fields in the structure: <pre>MQTMC2 MyTMC = {MQTMC2_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQTMC2 MQTMC2;
struct tagMQTMC2 {
    MQCHAR4    StrucId;        /* Structure identifier */
    MQCHAR4    Version;        /* Structure version number */
    MQCHAR48   QName;          /* Name of triggered queue */
    MQCHAR48   ProcessName;    /* Name of process object */
    MQCHAR64   TriggerData;    /* Trigger data */
    MQCHAR4    ApplType;       /* Application type */
    MQCHAR256  ApplId;         /* Application identifier */
    MQCHAR128  EnvData;        /* Environment data */
    MQCHAR128  UserData;       /* User data */
    MQCHAR48   QMgrName;       /* Queue manager name */
};
```

COBOL declaration

```
** MQTMC2 structure
10 MQTMC2.
** Structure identifier
15 MQTMC2-STRUCID PIC X(4).
** Structure version number
15 MQTMC2-VERSION PIC X(4).
** Name of triggered queue
15 MQTMC2-QNAME PIC X(48).
** Name of process object
15 MQTMC2-PROCESSNAME PIC X(48).
** Trigger data
15 MQTMC2-TRIGGERDATA PIC X(64).
** Application type
15 MQTMC2-APPLTYPE PIC X(4).
** Application identifier
15 MQTMC2-APPLID PIC X(256).
** Environment data
15 MQTMC2-ENVDATA PIC X(128).
** User data
15 MQTMC2-USERSDATA PIC X(128).
** Queue manager name
15 MQTMC2-QMGRNAME PIC X(48).
```

PL/I declaration

```
dcl
1 MQTMC2 based,
3 StrucId char(4), /* Structure identifier */
3 Version char(4), /* Structure version number */
3 QName char(48), /* Name of triggered queue */
3 ProcessName char(48), /* Name of process object */
3 TriggerData char(64), /* Trigger data */
3 ApplType char(4), /* Application type */
3 ApplId char(256), /* Application identifier */
3 EnvData char(128), /* Environment data */
```

```

3 UserData      char(128), /* User data */
3 QMgrName      char(48); /* Queue manager name */

```

High Level Assembler declaration

```

MQTMC          DSECT
MQTMC_STRUCID  DS   CL4    Structure identifier
MQTMC_VERSION  DS   CL4    Structure version number
MQTMC_QNAME    DS   CL48   Name of triggered queue
MQTMC_PROCESSNAME DS   CL48 Name of process object
MQTMC_TRIGGERDATA DS   CL64 Trigger data
MQTMC_APPLTYPE DS   CL4    Application type
MQTMC_APPLID   DS   CL256  Application identifier
MQTMC_ENVDATA  DS   CL128  Environment data
MQTMC_USERDATA DS   CL128  User data
MQTMC_QMGRNAME DS   CL48   Queue manager name
*
MQTMC_LENGTH   EQU   *-MQTMC
               ORG   MQTMC
MQTMC_AREA     DS    CL(MQTMC_LENGTH)

```

Visual Basic declaration

```

Type MQTMC2
  StrucId   As String*4   'Structure identifier'
  Version   As String*4   'Structure version number'
  QName     As String*48  'Name of triggered queue'
  ProcessName As String*48 'Name of process object'
  TriggerData As String*64 'Trigger data'
  ApplType  As String*4   'Application type'
  ApplId    As String*256 'Application identifier'
  EnvData   As String*128 'Environment data'
  UserData  As String*128 'User data'
  QMgrName  As String*48  'Queue manager name'
End Type

```

MQWIH - Work information header

The following table summarizes the fields in the structure.

Table 545. Fields in MQWIH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQWIH structure	StrucLength
<i>Encoding</i>	Numeric encoding of data that follows MQWIH	Encoding
<i>CodedCharSetId</i>	Character-set identifier of data that follows MQWIH	CodedCharSetId
<i>Format</i>	Format name of data that follows MQWIH	Format
<i>Flags</i>	Flags	Flags
<i>ServiceName</i>	Service name	ServiceName
<i>ServiceStep</i>	Service step name	ServiceStep
<i>MsgToken</i>	Message token	MsgToken
<i>Reserved</i>	Reserved	Reserved

Overview for MQWIH

The MQWIH structure describes the header information for a message sent to z/OS workload management (WLM). For any IBM MQ supported platform you can create and transmit a message that

includes the MQWIH structure, but only an IBM MQ for z/OS queue manager can interact with WLM. Therefore, for the message to get to WLM from a non-z/OS queue manager, your queue manager network must include at least one z/OS queue manager through which the message can be routed.

Availability: All IBM MQ systems, plus IBM MQ clients connected to these systems.

Purpose: The MQWIH structure describes the information that must be present at the start of a message that is to be handled by the z/OS workload manager.

Format name: MQFMT_WORK_INFO_HEADER.

Character set and encoding: The fields in the MQWIH structure are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes MQWIH, or by those fields in the MQMD structure if the MQWIH is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

Usage: If a message is to be processed by the z/OS workload manager, the message must begin with an MQWIH structure.

Fields for MQWIH

The MQWIH structure contains the following fields; the fields are described in **alphabetical order**:

CodedCharSetId (MQLONG)

This specifies the character set identifier of the data that follows the MQWIH structure; it does not apply to character data in the MQWIH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. You can use the following special value:

MQCCSI_INHERIT

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value MQCCSI_INHERIT is not returned by the MQGET call.

MQCCSI_INHERIT cannot be used if the value of the *PutApplType* field in MQMD is MQAT_BROKER.

The initial value of this field is MQCCSI_UNDEFINED.

Encoding (MQLONG)

This specifies the numeric encoding of the data that follows the MQWIH structure; it does not apply to numeric data in the MQWIH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

Flags (MQLONG)

The value must be:

MQWIH_NONE

No flags.

The initial value of this field is MQWIH_NONE.

Format (MQCHAR8)

This specifies the format name of the data that follows the MQWIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *Format* field in MQMD.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

MsgToken (MQBYTE16)

This is a message token that uniquely identifies the message.

For the MQPUT and MQPUT1 calls, this field is ignored. The length of this field is given by MQ_MSG_TOKEN_LENGTH. The initial value of this field is MQMTOK_NONE.

Reserved (MQCHAR32)

This is a reserved field; it must be blank.

ServiceName (MQCHAR32)

This is the name of the service that is to process the message.

The length of this field is given by MQ_SERVICE_NAME_LENGTH. The initial value of this field is 32 blank characters.

ServiceStep (MQCHAR8)

This is the name of the step of *ServiceName* to which the message relates.

The length of this field is given by MQ_SERVICE_STEP_LENGTH. The initial value of this field is 8 blank characters.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQWIH_STRUC_ID

Identifier for work information header structure.

For the C programming language, the constant MQWIH_STRUC_ID_ARRAY is also defined; this has the same value as MQWIH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQWIH_STRUC_ID.

StrucLength (MQLONG)

This is the length of the MQWIH structure. The value must be:

MQWIH_LENGTH_1

Length of version-1 work information header structure.

The following constant specifies the length of the current version:

MQWIH_CURRENT_LENGTH

Length of current version of work information header structure.

The initial value of this field is MQWIH_LENGTH_1.

Version (MQLONG)

This is the structure version number. The value must be:

MQWIH_VERSION_1

Version-1 work information header structure.

The following constant specifies the version number of the current version:

MQWIH_CURRENT_VERSION

Current version of work information header structure.

The initial value of this field is MQWIH_VERSION_1.

Initial values and language declarations for MQWIH

Table 546. Initial values of fields in MQWIH for MQWIH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQWIH_STRUC_ID	'WIH¬'
<i>Version</i>	MQWIH_VERSION_1	1
<i>StrucLength</i>	MQWIH_LENGTH_1	120
<i>Encoding</i>	None	0
<i>CodedCharSetId</i>	MQCCSI_UNDEFINED	0
<i>Format</i>	MQFMT_NONE	Blanks
<i>Flags</i>	MQWIH_NONE	0
<i>ServiceName</i>	None	Blanks
<i>ServiceStep</i>	None	Blanks
<i>MsgToken</i>	MQMTOK_NONE	Nulls
<i>Reserved</i>	None	Blanks
Notes: <ol style="list-style-type: none"> 1. The symbol ¬ represents a single blank character. 2. In the C programming language, the macro variable MQWIH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQWIH MyWIH = {MQWIH_DEFAULT};</pre> 		

C declaration

```
typedef struct tagMQWIH MQWIH;
struct tagMQWIH {
    MQCHAR4    StrucId;        /* Structure identifier */
    MQLONG     Version;        /* Structure version number */
    MQLONG     StrucLength;    /* Length of MQWIH structure */
    MQLONG     Encoding;       /* Numeric encoding of data that follows
                               MQWIH */
    MQLONG     CodedCharSetId; /* Character-set identifier of data that
                               follows MQWIH */
    MQCHAR8    Format;         /* Format name of data that follows
                               MQWIH */
    MQLONG     Flags;          /* Flags */
    MQCHAR32   ServiceName;    /* Service name */
    MQCHAR8    ServiceStep;    /* Service step name */
    MQBYTE16   MsgToken;       /* Message token */
    MQCHAR32   Reserved;       /* Reserved */
};
```

COBOL declaration

```
**  MQWIH structure
10 MQWIH.
**  Structure identifier
15 MQWIH-STRUCID      PIC X(4).
**  Structure version number
15 MQWIH-VERSION     PIC S9(9) BINARY.
**  Length of MQWIH structure
15 MQWIH-STRUCLength  PIC S9(9) BINARY.
**  Numeric encoding of data that follows MQWIH
15 MQWIH-ENCODING     PIC S9(9) BINARY.
**  Character-set identifier of data that follows MQWIH
15 MQWIH-CODEDCHARSETID PIC S9(9) BINARY.
**  Format name of data that follows MQWIH
```

```

15 MQWIH-FORMAT          PIC X(8).
**      Flags
15 MQWIH-FLAGS           PIC S9(9) BINARY.
**      Service name
15 MQWIH-SERVICENAME     PIC X(32).
**      Service step name
15 MQWIH-SERVICESTEP     PIC X(8).
**      Message token
15 MQWIH-MSGTOKEN        PIC X(16).
**      Reserved
15 MQWIH-RESERVED        PIC X(32).

```

PL/I declaration

```

dcl
1 MQWIH based,
3 StrucId          char(4),          /* Structure identifier */
3 Version          fixed bin(31), /* Structure version number */
3 StrucLength      fixed bin(31), /* Length of MQWIH structure */
3 Encoding         fixed bin(31), /* Numeric encoding of data that
                                follows MQWIH */
3 CodedCharSetId   fixed bin(31), /* Character-set identifier of data
                                that follows MQWIH */
3 Format           char(8),          /* Format name of data that follows
                                MQWIH */
3 Flags           fixed bin(31), /* Flags */
3 ServiceName     char(32),          /* Service name */
3 ServiceStep     char(8),          /* Service step name */
3 MsgToken       char(16),          /* Message token */
3 Reserved        char(32);         /* Reserved */

```

High Level Assembler declaration

```

MQWIH          DSECT
MQWIH_STRUCID  DS    CL4    Structure identifier
MQWIH_VERSION  DS    F      Structure version number
MQWIH_STRUCLNGTH DS    F      Length of MQWIH structure
MQWIH_ENCODING DS    F      Numeric encoding of data that follows
*              MQWIH
MQWIH_CODEDCHARSETID DS    F      Character-set identifier of data that
*              follows MQWIH
MQWIH_FORMAT   DS    CL8    Format name of data that follows MQWIH
MQWIH_FLAGS    DS    F      Flags
MQWIH_SERVICENAME DS    CL32  Service name
MQWIH_SERVICESTEP DS    CL8   Service step name
MQWIH_MSGTOKEN DS    XL16   Message token
MQWIH_RESERVED DS    CL32   Reserved
*
MQWIH_LENGTH   EQU    *-MQWIH
ORG    MQWIH
MQWIH_AREA     DS    CL(MQWIH_LENGTH)

```

Visual Basic declaration

```

Type MQWIH
    StrucId          As String*4    'Structure identifier'
    Version          As Long        'Structure version number'
    StrucLength      As Long        'Length of MQWIH structure'
    Encoding         As Long        'Numeric encoding of data that follows'
                                'MQWIH'
    CodedCharSetId   As Long        'Character-set identifier of data that'
                                'follows MQWIH'
    Format           As String*8    'Format name of data that follows MQWIH'
    Flags           As Long        'Flags'
    ServiceName     As String*32    'Service name'
    ServiceStep     As String*8     'Service step name'
    MsgToken       As MQBYTE16     'Message token'
    Reserved        As String*32    'Reserved'
End Type

```

MQXP - Exit parameter block

The following table summarizes the fields in the structure.

Table 547. Fields in MQXP

Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>ExitId</i>	Exit identifier	ExitId
<i>ExitReason</i>	Reason for invocation of exit	ExitReason
<i>ExitResponse</i>	Response from exit	ExitResponse
<i>ExitCommand</i>	API call code	ExitCommand
<i>ExitParmCount</i>	Parameter count	ExitParmCount
<i>ExitUserArea</i>	User area	ExitUserArea

Overview for MQXP

Availability: z/OS.

Purpose: The MQXP structure is used as an input/output parameter to the API-crossing exit. For more information about this exit, see [The API-crossing exit](#).

Character set and encoding: Character data in MQXP is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in MQXP is in the native machine encoding; this is given by MQENC_NATIVE.

Fields for MQXP

The MQXP structure contains the following fields; the fields are described in **alphabetical order**:

ExitCommand (MQLONG)

This field is set on entry to the exit routine. It identifies the API call that caused the exit to be invoked:

MQXC_CALLBACK

The CALLBACK call.

MQXC_MQBACK

The MQBACK call.

MQXC_MQCB

The MQCB call.

MQXC_MQCLOSE

The MQCLOSE call.

MQXC_MQCMIT

The MQCMIT call.

MQXC_MQCTL

The MQCTL call.

MQXC_MQGET

The MQGET call.

MQXC_MQINQ

The MQINQ call.

MQXC_MQOPEN

The MQOPEN call.

MQXC_MQPUT

The MQPUT call.

MQXC_MQPUT1

The MQPUT1 call.

MQXC_MQSET

The MQSET call.

MQXC_MQSTAT

The MQSTAT call.

MQXC_MQSUB

The MQSUB call.

MQXC_MQSUBRQ

The MQSUBRQ call.

This is an input field to the exit.

ExitId (MQLONG)

This is set on entry to the exit routine, and indicates the type of exit:

MQXT_API_CROSSING_EXIT

API-crossing exit for CICS.

This is an input field to the exit.

ExitParmCount (MQLONG)

This field is set on entry to the exit routine. It contains the number of parameters that the MQ call takes. These are:

Call name	Number of parameters
MQBACK	3
MQCLOSE	5
MQCMIT	3
MQGET	9
MQINQ	10
MQOPEN	6
MQPUT	8
MQPUT1	8
MQSET	10

This is an input field to the exit.

ExitReason (MQLONG)

This is set on entry to the exit routine. For the API-crossing exit it indicates whether the routine is called before or after execution of the API call:

MQXR_BEFORE

Before API execution.

MQXR_AFTER

After API execution.

This is an input field to the exit.

ExitResponse (MQLONG)

The value is set by the exit to communicate with the caller. The following values are defined:

MQXCC_OK

Exit completed successfully.

MQXCC_SUPPRESS_FUNCTION

Suppress function.

When this value is set by an API-crossing exit called *before* the API call, the API call is not performed. The *CompCode* for the call is set to MQCC_FAILED, the *Reason* is set to MQRC_SUPPRESSED_BY_EXIT, and all other parameters remain as the exit left them.

When this value is set by an API-crossing exit called *after* the API call, it is ignored by the queue manager.

MQXCC_SKIP_FUNCTION

Skip function.

When this value is set by an API-crossing exit called *before* the API call, the API call is not performed; the *CompCode* and *Reason* and all other parameters remain as the exit left them.

When this value is set by an API-crossing exit called *after* the API call, it is ignored by the queue manager.

This is an output field from the exit.

ExitUserArea (MQBYTE16)

This is a field that is available for the exit to use. It is initialized to binary zero for the length of the field before the first invocation of the exit for the task, and thereafter any changes made to this field by the exit are preserved across invocations of the exit. The following value is defined:

MQXUA_NONE

No user information.

The value is binary zero for the length of the field.

For the C programming language, the constant MQXUA_NONE_ARRAY is also defined; this has the same value as MQXUA_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH. This is an input/output field to the exit.

Reserved (MQLONG)

This is a reserved field. Its value is not significant to the exit.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQXP_STRUC_ID

Identifier for exit parameter structure.

For the C programming language, the constant MQXP_STRUC_ID_ARRAY is also defined; this has the same value as MQXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)

This is the structure version number. The value must be:

MQXP_VERSION_1

Version number for exit parameter-block structure.

Note: When a new version of this structure is introduced, the layout of the existing part is not changed. The exit must therefore check that the version number is equal to or greater than the lowest version that contains the fields that the exit needs to use.

This is an input field to the exit.

Language declarations

This structure is supported in the following programming languages.

C declaration

```
typedef struct tagMQXP MQXP;
struct tagMQXP {
    MQCHAR4    StrucId;        /* Structure identifier */
    MQLONG     Version;        /* Structure version number */
    MQLONG     ExitId;         /* Exit identifier */
    MQLONG     ExitReason;     /* Reason for invocation of exit */
    MQLONG     ExitResponse;   /* Response from exit */
    MQLONG     ExitCommand;    /* API call code */
    MQLONG     ExitParmCount;  /* Parameter count */
    MQLONG     Reserved;       /* Reserved */
    MQBYTE16   ExitUserArea;   /* User area */
};
```

COBOL declaration

```
**      MQXP structure
10 MQXP.
**      Structure identifier
15 MQXP-STRUCID      PIC X(4).
**      Structure version number
15 MQXP-VERSION      PIC S9(9) BINARY.
**      Exit identifier
15 MQXP-EXITID       PIC S9(9) BINARY.
**      Reason for invocation of exit
15 MQXP-EXITREASON   PIC S9(9) BINARY.
**      Response from exit
15 MQXP-EXITRESPONSE PIC S9(9) BINARY.
**      API call code
15 MQXP-EXITCOMMAND  PIC S9(9) BINARY.
**      Parameter count
15 MQXP-EXITPARMCOUNT PIC S9(9) BINARY.
**      Reserved
15 MQXP-RESERVED     PIC S9(9) BINARY.
**      User area
15 MQXP-EXITUSERAREA PIC X(16).
```

PL/I declaration

```
dcl
1 MQXP based,
3 StrucId      char(4),        /* Structure identifier */
3 Version      fixed bin(31), /* Structure version number */
3 ExitId       fixed bin(31), /* Exit identifier */
3 ExitReason   fixed bin(31), /* Reason for invocation of exit */
3 ExitResponse fixed bin(31), /* Response from exit */
3 ExitCommand  fixed bin(31), /* API call code */
3 ExitParmCount fixed bin(31), /* Parameter count */
3 Reserved     fixed bin(31), /* Reserved */
3 ExitUserArea char(16);      /* User area */
```

High Level Assembler declaration

```
MQXP          DSECT
MQXP_STRUCID   DS    CL4    Structure identifier
MQXP_VERSION   DS    F      Structure version number
MQXP_EXITID    DS    F      Exit identifier
MQXP_EXITREASON DS    F      Reason for invocation of exit
MQXP_EXITRESPONSE DS    F      Response from exit
MQXP_EXITCOMMAND DS    F      API call code
MQXP_EXITPARMCOUNT DS    F      Parameter count
MQXP_RESERVED  DS    F      Reserved
MQXP_EXITUSERAREA DS    XL16 User area
*
MQXP_LENGTH    EQU    *-MQXP
ORG    MQXP
MQXP_AREA      DS     CL(MQXP_LENGTH)
```

MQXQH - Transmission-queue header

The following table summarizes the fields in the structure.

Table 548. Fields in MQXQH		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>RemoteQName</i>	Name of destination queue	RemoteQName
<i>RemoteQMgrName</i>	Name of destination queue manager	RemoteQMgrName
<i>MsgDesc</i>	Original message descriptor	MsgDesc

Overview for MQXQH

Availability: All IBM MQ systems and IBM MQ clients.

Purpose: The MQXQH structure describes the information that is prefixed to the application message data of messages when they are on transmission queues. A transmission queue is a special type of local queue that temporarily holds messages destined for remote queues (that is, destined for queues that do not belong to the local queue manager). A transmission queue is denoted by the *Usage* queue attribute having the value MQUS_TRANSMISSION.

Format name: MQFMT_XMIT_Q_HEADER.

Character set and encoding: Data in MQXQH must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by MQENC_NATIVE.

Set the character set and encoding of the MQXQH into the *CodedCharSetId* and *Encoding* fields in:

- The separate MQMD (if the MQXQH structure is at the start of the message data), or
- The header structure that precedes the MQXQH structure (all other cases).

Usage: A message that is on a transmission queue has *two* message descriptors:

- One message descriptor is stored separately from the message data; this is called the *separate message descriptor*, and is generated by the queue manager when the message is placed on the transmission queue. Some of the fields in the separate message descriptor are copied from the message descriptor provided by the application on the MQPUT or MQPUT1 call.

The separate message descriptor is the one that is returned to the application in the *MsgDesc* parameter of the MQGET call when the message is removed from the transmission queue.

- A second message descriptor is stored within the MQXQH structure as part of the message data; this is called the *embedded message descriptor*, and is a copy of the message descriptor that was provided by the application on the MQPUT or MQPUT1 call (with minor variations).

The embedded message descriptor is always a version-1 MQMD. If the message put by the application has nondefault values for one or more of the version-2 fields in the MQMD, an MQMDE structure follows the MQXQH, and is in turn followed by the application message data (if any). The MQMDE is either:

- Generated by the queue manager (if the application uses a version-2 MQMD to put the message), or
- Already present at the start of the application message data (if the application uses a version-1 MQMD to put the message).

The embedded message descriptor is the one that is returned to the application in the *MsgDesc* parameter of the MQGET call when the message is removed from the final destination queue.

Fields in the separate message descriptor: The fields in the separate message descriptor are set by the queue manager as shown. If the queue manager does not support the version-2 MQMD, a version-1 MQMD is used without loss of function.

Field in separate MQMD	Value used
<i>StrucId</i>	MQMD_STRUC_ID
<i>Version</i>	MQMD_VERSION_2
<i>Report</i>	Copied from the embedded message descriptor, but with the bits identified by MQRO_ACCEPT_UNSUP_IF_XMIT_MASK set to zero. (This prevents a COA or COD report message being generated when a message is placed on or removed from a transmission queue.)
<i>MsgType</i>	Copied from the embedded message descriptor.
<i>Expiry</i>	Copied from the embedded message descriptor.
<i>Feedback</i>	Copied from the embedded message descriptor.
<i>Encoding</i>	MQENC_NATIVE (see note)
<i>CodedCharSetId</i>	Queue manager's <i>CodedCharSetId</i> attribute.
<i>Format</i>	MQFMT_XMIT_Q_HEADER
<i>Priority</i>	Copied from the embedded message descriptor.
<i>Persistence</i>	Copied from the embedded message descriptor.
<i>MsgId</i>	A new value is generated by the queue manager. This message identifier is different from the <i>MsgId</i> that the queue manager may have generated for the embedded message descriptor described previously.
<i>CorrelId</i>	The <i>MsgId</i> from the embedded message descriptor. For messages being put to the SYSTEM.CLUSTER.TRANSMIT.QUEUE, <i>CorrelId</i> is reserved for internal use.
<i>BackoutCount</i>	0
<i>ReplyToQ</i>	Copied from the embedded message descriptor.
<i>ReplyToQMgr</i>	Copied from the embedded message descriptor.
<i>UserIdentifier</i>	Copied from the embedded message descriptor.
<i>AccountingToken</i>	Copied from the embedded message descriptor. For messages being put to the SYSTEM.CLUSTER.TRANSMIT.QUEUE, <i>AccountingToken</i> is reserved for internal use.
<i>ApplIdentityData</i>	Copied from the embedded message descriptor.
<i>PutApplType</i>	MQAT_QMGR
<i>PutApplName</i>	First 28 bytes of the queue-manager name.
<i>PutDate</i>	Date when message was put on transmission queue.
<i>PutTime</i>	Time when message was put on transmission queue.
<i>ApplOriginData</i>	Blanks
<i>GroupId</i>	MQGI_NONE
<i>MsgSeqNumber</i>	1
<i>Offset</i>	0
<i>MsgFlags</i>	MQMF_NONE
<i>OriginalLength</i>	MQOL_UNDEFINED

- On Windows, the value of MQENC_NATIVE for Micro Focus COBOL differs from the value for C. The value in the *Encoding* field in the separate message descriptor is always the value for C in these

environments; this value is 546 in decimal. Also, the integer fields in the MQXQH structure are in the encoding that corresponds to this value (the native Intel encoding).

Fields in the embedded message descriptor: The fields in the embedded message descriptor have the same values as those in the *MsgDesc* parameter of the MQPUT or MQPUT1 call, except for the following:

- The *Version* field always has the value MQMD_VERSION_1.
- If the *Priority* field has the value MQPRI_PRIORITY_AS_Q_DEF, it is replaced by the value of the queue's *DefPriority* attribute.
- If the *Persistence* field has the value MQPER_PERSISTENCE_AS_Q_DEF, it is replaced by the value of the queue's *DefPersistence* attribute.
- If the *MsgId* field has the value MQMI_NONE, or the MQPMO_NEW_MSG_ID option was specified, or the message is a distribution-list message, *MsgId* is replaced by a new message identifier generated by the queue manager.

When a distribution-list message is split into smaller distribution-list messages placed on different transmission queues, the *MsgId* field in each of the new embedded message descriptors is the same as that in the original distribution-list message.

- If the MQPMO_NEW_CORREL_ID option was specified, *CorrelId* is replaced by a new correlation identifier generated by the queue manager.
- The context fields are set as indicated by the MQPMO_*_CONTEXT options specified in the *PutMsgOpts* parameter; the context fields are:
 - *AccountingToken*
 - *ApplIdentityData*
 - *ApplOriginData*
 - *PutApplName*
 - *PutApplType*
 - *PutDate*
 - *PutTime*
 - *UserIdentifier*
- The version-2 fields (if they were present) are removed from the MQMD, and moved into an MQMDE structure, if one or more of the version-2 fields has a nondefault value.

Putting messages on remote queues: When an application puts a message on a remote queue (either by specifying the name of the remote queue directly, or by using a local definition of the remote queue), the local queue manager:

- Creates an MQXQH structure containing the embedded message descriptor
- Appends an MQMDE if one is needed and is not already present
- Appends the application message data
- Places the message on an appropriate transmission queue

Putting messages directly on transmission queues: An application can also put a message directly on a transmission queue. In this case the application must prefix the application message data with an MQXQH structure, and initialize the fields with appropriate values. In addition, the *Format* field in the *MsgDesc* parameter of the MQPUT or MQPUT1 call must have the value MQFMT_XMIT_Q_HEADER.

Character data in the MQXQH structure created by the application must be in the character set of the local queue manager (defined by the *CodedCharSetId* queue-manager attribute), and integer data must be in the native machine encoding. In addition, character data in the MQXQH structure must be padded with blanks to the defined length of the field; the data must not be ended prematurely by using a null character, because the queue manager does not convert the null and subsequent characters to blanks in the MQXQH structure.

However, the queue manager does not check that an MQXQH structure is present, or that valid values have been specified for the fields.

Applications should not put their messages directly to the SYSTEM.CLUSTER.TRANSMIT.QUEUE.

Getting messages from transmission queues: Applications that get messages from a transmission queue must process the information in the MQXQH structure in an appropriate fashion. The presence of the MQXQH structure at the beginning of the application message data is indicated by the value MQFMT_XMIT_Q_HEADER being returned in the *Format* field in the *MsgDesc* parameter of the MQGET call. The values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter indicate the character set and encoding of the character and integer data in the MQXQH structure. The character set and encoding of the application message data are defined by the *CodedCharSetId* and *Encoding* fields in the embedded message descriptor.

Fields for MQXQH

The MQXQH structure contains the following fields; the fields are described in **alphabetical order**:

MsgDesc (MQMD1)

This is the embedded message descriptor, and is a close copy of the message descriptor MQMD that was specified as the *MsgDesc* parameter on the MQPUT or MQPUT1 call when the message was originally put to the remote queue.

Note: This is a version-1 MQMD.

The initial values of the fields in this structure are the same as those in the MQMD structure.

RemoteQMgrName (MQCHAR48)

This is the name of the queue manager or queue-sharing group that owns the queue that is the apparent eventual destination for the message.

If the message is a distribution-list message, *RemoteQMgrName* is blank.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

RemoteQName (MQCHAR48)

This is the name of the message queue that is the apparent eventual destination for the message (this might prove not to be the eventual destination if, for example, this queue is defined at *RemoteQMgrName* to be a local definition of another remote queue).

If the message is a distribution-list message (that is, the *Format* field in the embedded message descriptor is MQFMT_DIST_HEADER), *RemoteQName* is blank.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

StrucId (MQCHAR4)

This is the structure identifier. The value must be:

MQXQH_STRUC_ID

Identifier for transmission-queue header structure.

For the C programming language, the constant MQXQH_STRUC_ID_ARRAY is also defined; this has the same value as MQXQH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQXQH_STRUC_ID.

Version (MQLONG)

This is the structure version number. The value must be:

MQXQH_VERSION_1

Version number for transmission-queue header structure.

The following constant specifies the version number of the current version:

MQXQH_CURRENT_VERSION

Current version of transmission-queue header structure.

The initial value of this field is MQXQH_VERSION_1.

Initial values and language declarations for MQXQH

Table 549. Initial values of fields in MQXQH for MQXQH		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQXQH_STRUC_ID	'XQH~'
<i>Version</i>	MQXQH_VERSION_1	1
<i>RemoteQName</i>	None	Null string or blanks
<i>RemoteQMgrName</i>	None	Null string or blanks
<i>MsgDesc</i>	Same names and values as MQMD; see Table 504 on page 443	-
Notes: 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 3. In the C programming language, the macro variable MQXQH_DEFAULT contains the values that are listed in the table. Use it in the following way to provide initial values for the fields in the structure: <pre>MQXQH MyXQH = {MQXQH_DEFAULT};</pre>		

C declaration

```
typedef struct tagMQXQH MQXQH;  
struct tagMQXQH {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    MQCHAR48   RemoteQName;      /* Name of destination queue */  
    MQCHAR48   RemoteQMgrName;   /* Name of destination queue manager */  
    MQMD1      MsgDesc;          /* Original message descriptor */  
};
```

COBOL declaration

```
**  MQXQH structure  
10 MQXQH.  
**    Structure identifier  
15 MQXQH-STRUCID          PIC X(4).  
**    Structure version number  
15 MQXQH-VERSION          PIC S9(9) BINARY.  
**    Name of destination queue  
15 MQXQH-REMOTEQNAME      PIC X(48).  
**    Name of destination queue manager  
15 MQXQH-REMOTEQMGRNAME   PIC X(48).  
**    Original message descriptor  
15 MQXQH-MSGDESC.  
**    Structure identifier  
20 MQXQH-MSGDESC-STRUCID  PIC X(4).  
**    Structure version number  
20 MQXQH-MSGDESC-VERSION  PIC S9(9) BINARY.  
**    Report options  
20 MQXQH-MSGDESC-REPORT   PIC S9(9) BINARY.  
**    Message type  
20 MQXQH-MSGDESC-MSGTYPE  PIC S9(9) BINARY.  
**    Expiry time  
20 MQXQH-MSGDESC-EXPIRY   PIC S9(9) BINARY.  
**    Feedback or reason code
```

```

20 MQXQH-MSGDESC-FEEDBACK          PIC S9(9) BINARY.
** Numeric encoding of message data
20 MQXQH-MSGDESC-ENCODING          PIC S9(9) BINARY.
** Character set identifier of message data
20 MQXQH-MSGDESC-CODEDCHARSETID    PIC S9(9) BINARY.
** Format name of message data
20 MQXQH-MSGDESC-FORMAT            PIC X(8).
** Message priority
20 MQXQH-MSGDESC-PRIORITY          PIC S9(9) BINARY.
** Message persistence
20 MQXQH-MSGDESC-PERSISTENCE       PIC S9(9) BINARY.
** Message identifier
20 MQXQH-MSGDESC-MSGID             PIC X(24).
** Correlation identifier
20 MQXQH-MSGDESC-CORRELID          PIC X(24).
** Backout counter
20 MQXQH-MSGDESC-BACKOUTCOUNT     PIC S9(9) BINARY.
** Name of reply-to queue
20 MQXQH-MSGDESC-REPLYTOQ          PIC X(48).
** Name of reply queue manager
20 MQXQH-MSGDESC-REPLYTOQMGR       PIC X(48).
** User identifier
20 MQXQH-MSGDESC-USERIDENTIFIER    PIC X(12).
** Accounting token
20 MQXQH-MSGDESC-ACCOUNTINGTOKEN   PIC X(32).
** Application data relating to identity
20 MQXQH-MSGDESC-APPLIDENTITYDATA  PIC X(32).
** Type of application that put the message
20 MQXQH-MSGDESC-PUTAPPLTYPE       PIC S9(9) BINARY.
** Name of application that put the message
20 MQXQH-MSGDESC-PUTAPPLNAME       PIC X(28).
** Date when message was put
20 MQXQH-MSGDESC-PUTDATE           PIC X(8).
** Time when message was put
20 MQXQH-MSGDESC-PUTTIME           PIC X(8).
** Application data relating to origin
20 MQXQH-MSGDESC-APPLORIGINDATA    PIC X(4).

```

PL/I declaration

```

dcl
  1 MQXQH based,
    3 StrucId          char(4),          /* Structure identifier */
    3 Version          fixed bin(31),    /* Structure version number */
    3 RemoteQName      char(48),         /* Name of destination queue */
    3 RemoteQMgrName   char(48),         /* Name of destination queue
                                          manager */
    3 MsgDesc,         /* Original message descriptor */
    5 StrucId          char(4),          /* Structure identifier */
    5 Version          fixed bin(31),    /* Structure version number */
    5 Report           fixed bin(31),    /* Report options */
    5 MsgType          fixed bin(31),    /* Message type */
    5 Expiry           fixed bin(31),    /* Expiry time */
    5 Feedback         fixed bin(31),    /* Feedback or reason code */
    5 Encoding         fixed bin(31),    /* Numeric encoding of message
                                          data */
    5 CodedCharSetId   fixed bin(31),    /* Character set identifier of
                                          message data */
    5 Format            char(8),          /* Format name of message data */
    5 Priority          fixed bin(31),    /* Message priority */
    5 Persistence      fixed bin(31),    /* Message persistence */
    5 MsgId            char(24),         /* Message identifier */
    5 CorrelId         char(24),         /* Correlation identifier */
    5 BackoutCount     fixed bin(31),    /* Backout counter */
    5 ReplyToQ         char(48),         /* Name of reply-to queue */
    5 ReplyToQMGR      char(48),         /* Name of reply queue manager */
    5 UserIdentifier   char(12),         /* User identifier */
    5 AccountingToken  char(32),         /* Accounting token */
    5 ApplIdentityData char(32),         /* Application data relating to
                                          identity */
    5 PutApplType      fixed bin(31),    /* Type of application that put the
                                          message */
    5 PutApplName      char(28),         /* Name of application that put the
                                          message */
    5 PutDate          char(8),          /* Date when message was put */
    5 PutTime          char(8),          /* Time when message was put */
    5 ApplOriginData   char(4);          /* Application data relating to
                                          origin */

```

High Level Assembler declaration

MQXQH	DSECT	
MQXQH_STRUCID	DS	CL4 Structure identifier
MQXQH_VERSION	DS	F Structure version number
MQXQH_REMOTEQNAME	DS	CL48 Name of destination queue
MQXQH_REMOTEQMGRNAME	DS	CL48 Name of destination queue manager
*		
MQXQH_MSGDESC	DS	0F Force fullword alignment
MQXQH_MSGDESC_STRUCID	DS	CL4 Structure identifier
MQXQH_MSGDESC_VERSION	DS	F Structure version number
MQXQH_MSGDESC_REPORT	DS	F Report options
MQXQH_MSGDESC_MSGTYPE	DS	F Message type
MQXQH_MSGDESC_EXPIRY	DS	F Expiry time
MQXQH_MSGDESC_FEEDBACK	DS	F Feedback or reason code
MQXQH_MSGDESC_ENCODING	DS	F Numeric encoding of message data
*		
MQXQH_MSGDESC_CODEDCHARSETID	DS	F Character set identifier of message data
*		
MQXQH_MSGDESC_FORMAT	DS	CL8 Format name of message data
MQXQH_MSGDESC_PRIORITY	DS	F Message priority
MQXQH_MSGDESC_PERSISTENCE	DS	F Message persistence
MQXQH_MSGDESC_MSGID	DS	XL24 Message identifier
MQXQH_MSGDESC_CORRELID	DS	XL24 Correlation identifier
MQXQH_MSGDESC_BACKOUTCOUNT	DS	F Backout counter
MQXQH_MSGDESC_REPLYTOQ	DS	CL48 Name of reply-to queue
MQXQH_MSGDESC_REPLYTOQMGR	DS	CL48 Name of reply queue manager
MQXQH_MSGDESC_USERIDENTIFIER	DS	CL12 User identifier
MQXQH_MSGDESC_ACCOUNTINGTOKEN	DS	XL32 Accounting token
MQXQH_MSGDESC_APPLIDENTITYDATA	DS	CL32 Application data relating to identity
*		
MQXQH_MSGDESC_PUTAPPLTYPE	DS	F Type of application that put the message
*		
MQXQH_MSGDESC_PUTAPPLNAME	DS	CL28 Name of application that put the message
*		
MQXQH_MSGDESC_PUTDATE	DS	CL8 Date when message was put
MQXQH_MSGDESC_PUTTIME	DS	CL8 Time when message was put
MQXQH_MSGDESC_APPLORIGINDATA	DS	CL4 Application data relating to origin
*		
MQXQH_MSGDESC_LENGTH	EQU	*-MQXQH_MSGDESC
	ORG	MQXQH_MSGDESC
MQXQH_MSGDESC_AREA	DS	CL(MQXQH_MSGDESC_LENGTH)
*		
MQXQH_LENGTH	EQU	*-MQXQH
	ORG	MQXQH
MQXQH_AREA	DS	CL(MQXQH_LENGTH)

Visual Basic declaration

```
Type MQXQH
  StrucId      As String*4 'Structure identifier'
  Version      As Long     'Structure version number'
  RemoteQName  As String*48 'Name of destination queue'
  RemoteQMgrName As String*48 'Name of destination queue manager'
  MsgDesc      As MQMD1    'Original message descriptor'
End Type
```

Function calls

This section gives information on all of the MQI calls that are possible. Descriptions, syntax, parameter information, usage notes, and language invocations for each possible language are given for each of the different calls.

Call descriptions

This section describes MQI calls.

- [“MQBACK - Back out changes” on page 601](#)
- [“MQBEGIN - Begin unit of work” on page 604](#)
- [“MQBUFMH - Convert buffer into message handle” on page 608](#)
- [“MQCB - Manage callback” on page 611](#)

- [“MQCB_FUNCTION - Callback function” on page 621](#)
- [“MQCLOSE - Close object” on page 622](#)
- [“MQCMIT - Commit changes” on page 630](#)
- [“MQCONN - Connect queue manager” on page 634](#)
- [“MQCONNX - Connect queue manager \(extended\)” on page 642](#)
- [“MQCRTMH - Create message handle” on page 647](#)
- [“MQCTL - Control callbacks” on page 650](#)
- [“MQDISC - Disconnect queue manager” on page 657](#)
- [“MQDLTMH - Delete message handle” on page 660](#)
- [“MQDLTMP - Delete message property” on page 663](#)
- [“MQGET - Get message” on page 665](#)
- [“MQINQ - Inquire object attributes” on page 677](#)
- [“MQINQMP - Inquire message property” on page 694](#)
- [“MQMHBUF - Convert message handle into buffer” on page 699](#)
- [“MQOPEN - Open object” on page 703](#)
- [“MQPUT - Put message” on page 720](#)
- [“MQPUT1 - Put one message” on page 734](#)
- [“MQSET - Set object attributes” on page 744](#)
- [“MQSETMP - Set message property” on page 750](#)
- [“MQSTAT - Retrieve status information” on page 754](#)
- [“MQMHBUF - Convert message handle into buffer” on page 699](#)
- [“MQSUB - Register subscription” on page 758](#)
- [“MQSUBRQ - Subscription request” on page 765](#)

Online help on the UNIX platforms, in the form of *man* pages, is available for these calls.

Note: The calls associated with data conversion, MQXCNVC and MQ_DATA_CONV_EXIT, are in [“Data-conversion exit” on page 870](#).

Conventions used in the call descriptions

For each call, this collection of topics gives a description of the parameters and usage of the call in a format that is independent of programming language. This is followed by typical invocations of the call, and typical declarations of its parameters, in each of the supported programming languages.

Important: When coding IBM MQ API calls you must ensure that all relevant parameters (as described in the following sections) are provided. Failure to do so can produce unpredictable results.

The description of each call contains the following sections:

Call name

The call name, followed by a brief description of the purpose of the call.

Parameters

For each parameter, the name is followed by its data type in parentheses () and one of the following:

input

You supply information in the parameter when you make the call.

output

The queue manager returns information in the parameter when the call completes or fails.

input/output

You supply information in the parameter when you make the call, and the queue manager changes the information when the call completes or fails.

For example:

Compcode (MQLONG) - output

In some cases, the data type is a structure. In all cases, there is more information about the data type or structure in [“Elementary data types” on page 231](#).

The last two parameters in each call are a completion code and a reason code. The completion code indicates whether the call completed successfully, partially, or not at all. Further information about the partial success or the failure of the call is given in the reason code. For more information about each completion and reason code, see [“Return codes” on page 838](#).

Usage notes

Additional information about the call, describing how to use it and any restrictions on its use.

Assembler language invocation

Typical invocation of the call, and declaration of its parameters, in assembler language.

C invocation

Typical invocation of the call, and declaration of its parameters, in C.

COBOL invocation

Typical invocation of the call, and declaration of its parameters, in COBOL.

PL/I invocation

Typical invocation of the call, and declaration of its parameters, in PL/I.

All parameters are passed by reference.

Visual Basic invocation

Typical invocation of the call, and declaration of its parameters, in Visual Basic.

Other notation conventions are:

Constants

Names of constants are shown in uppercase; for example, MQOO_OUTPUT. A set of constants having the same prefix is shown as follows: MQIA_*. See [“Constants” on page 59](#) for the value of a constant.

Arrays

In some calls, parameters are arrays of character strings that do not have fixed sizes. In the descriptions of these parameters, a lowercase *n* represents a numeric constant. When you code the declaration for that parameter, replace the *n* with the numeric value that you require.

Using the calls in the C language

Parameters that are *input only* and of type MQHCONN, MQHOBJ, MQHMSG, or MQLONG are passed by value. For all other parameters, the *address* of the parameter is passed by value.

You do not need to specify all parameters that are passed by address every time that you invoke a function. Where you do not need a particular parameter, specify a null pointer as the parameter on the function invocation, in place of the address of parameter data. Parameters for which this is possible are identified in the call descriptions.

No parameter is returned as the value of the call; in C terminology, this means that all calls return void .

Declaring the Buffer parameter

The MQGET , MQPUT , and MQPUT1 calls each have one parameter that has an undefined data type: the *Buffer* parameter. Use this parameter to send and receive the application's message data.

Parameters of this sort are shown in the C examples as arrays of MQBYTE. You can declare the parameters in this way, but it is usually more convenient to declare them as the particular structure that describes the layout of the data in the message. The function prototype declares the parameter as a pointer-to-void, so that you can specify the address of any sort of data as the parameter on the call invocation.

Pointer-to-void is a pointer to data of undefined format. It is defined as:


```
typedef void *PMQVOID;
```

MQBACK - Back out changes

The MQBACK call indicates to the queue manager that all the message gets and puts that have occurred since the last sync point are to be backed out.

Messages put as part of a unit of work are deleted; messages retrieved as part of a unit of work are reinstated on the queue.

- On z/OS, this call is used only by batch programs (including IMS batch DL/I programs).
- On IBM i, this call is not supported for applications running in compatibility mode.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Syntax

MQBACK (*Hconn*, *Compcode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

Compcode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_OUTCOME_PENDING

(2124, X'84C') Result of back-out operation is pending.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged.

MQRC_OUTCOME_MIXED

(2123, X'84B') Result of commit or back-out operation is mixed.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#)

Usage notes

1. You can use this call only when the queue manager itself coordinates the unit of work. This can be:
 - A local unit of work, where the changes affect only MQ resources.
 - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting MQ resources.

For further details about local and global units of work, see [“MQBEGIN - Begin unit of work” on page 604](#).

2. In environments where the queue manager does not coordinate the unit of work, use the appropriate back-out call instead of MQBACK. The environment might also support an implicit back out caused by the application terminating abnormally.
 - On z/OS, use the following calls:
 - Batch programs (including IMS batch DL/I programs) can use the MQBACK call if the unit of work affects only MQ resources. However, if the unit of work affects both MQ resources and resources belonging to other resource managers (for example, Db2®), use the SRRBACK call provided by the z/OS Recoverable Resource Service (RRS). The SRRBACK call backs out changes to resources belonging to the resource managers that have been enabled for RRS coordination.
 - CICS applications must use the EXEC CICS SYNCPOINT ROLLBACK command to back out the unit of work. Do not use the MQBACK call for CICS applications.
 - IMS applications (other than batch DL/I programs) must use IMS calls such as ROLB to back out the unit of work. Do not use the MQBACK call for IMS applications (other than batch DL/I programs).

- On IBM i, use this call for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
3. If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager”](#) on page 657 for further details.
 4. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

The queue manager keeps *three* sets of group and segment information, one set for each of the following:

- The last successful MQPUT call (this can be part of a unit of work).
 - The last successful MQGET call that removed a message from the queue (this can be part of a unit of work).
 - The last successful MQGET call that browsed a message on the queue (this *cannot* be part of a unit of work).
5. The information associated with the MQGET call is restored to the value that it had before the first successful MQGET call for that queue handle in the current unit of work.

Queues that were updated by the application after the unit of work started, but outside the scope of the unit of work, do not have their group and segment information restored if the unit of work is backed out.

Restoring the group and segment information to its previous value when a unit of work is backed out allows the application to spread a large message group or large logical message consisting of many segments across several units of work, and to restart at the correct point in the message group or logical message if one of the units of work fails.

Using several units of work might be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point if a system failure occurs.

For details of how to restart at the correct point after a system failure, see the MQPMO_LOGICAL_ORDER option described in [“MQPMO - Put-message options”](#) on page 475, and the MQGMO_LOGICAL_ORDER option described in [“MQGMO - Get-message options”](#) on page 350.

The remaining usage notes apply only when the queue manager coordinates the units of work.

6. A unit of work has the same scope as a connection handle. All MQ calls that affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *Hconn* parameter described in [“MQCONN - Connect queue manager”](#) on page 634 for information about the scope of connection handles.
7. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
8. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but that never issues a commit or backout call, can fill queues with messages that are not available to other applications. To guard against this possibility, the administrator must set the *MaxUncommittedMsgs* queue-manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.

C invocation

```
MQBACK (Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;      /* Connection handle */
MQLONG   CompCode;   /* Completion code */
MQLONG   Reason;     /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQBACK' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.
```

PL/I invocation

```
call MQBACK (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQBACK,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN      DS  F  Connection handle
COMPCODE   DS  F  Completion code
REASON     DS  F  Reason code qualifying COMPCODE
```

Visual Basic invocation

```
MQBACK Hconn, CompCode, Reason
```

Declare the parameters as follows:

```
Dim Hconn      As Long 'Connection handle'
Dim CompCode   As Long 'Completion code'
Dim Reason     As Long 'Reason code qualifying CompCode'
```

MQBEGIN - Begin unit of work

The MQBEGIN call begins a unit of work that is coordinated by the queue manager, and that can involve external resource managers.

Syntax

```
MQBEGIN (Hconn, BeginOptions, Compcode, Reason)
```

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

Hconn must be a nonshared connection handle. If a shared connection handle is specified, the call fails with reason code MQRC_HCONN_ERROR. See the description of the MQCNO_HANDLE_SHARE_* options in [“MQCNO - Connect options” on page 306](#) for more information about shared and nonshared handles.

BeginOptions

Type: MQBO - input/output

These are options that control the action of MQBEGIN, as described in [“MQBO - Begin options” on page 269](#).

If no options are required, programs written in C or S/390 assembler can specify a null parameter address, instead of specifying the address of an MQBO structure.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_NO_EXTERNAL_PARTICIPANTS

(2121, X'849') No participating resource managers registered.

MQRC_PARTICIPANT_NOT_AVAILABLE

(2122, X'84A') Participating resource manager not available.

If *CompCode* is MQCC_FAILED:

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_BO_ERROR

(2134, X'856') Begin-options structure not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UOW_IN_PROGRESS

(2128, X'850') Unit of work already started.

For more information about these reason codes, see [Messages and reason codes](#).

Usage notes

1. Use the MQBEGIN call to start a unit of work that is coordinated by the queue manager and that might involve changes to resources owned by other resource managers. The queue manager supports three types of unit-of-work:
 - **Queue-manager-coordinated local unit of work:** A unit of work in which the queue manager is the only resource manager participating, and so the queue manager acts as the unit-of-work coordinator.
 - To start this type of unit of work, specify the MQPMO_SYNCPOINT or MQGMO_SYNCPOINT option on the first MQPUT, MQPUT1, or MQGET call in the unit of work.
 - To commit or back out this type of unit of work, use the MQCMIT or MQBACK call.
 - **Queue-manager-coordinated global unit of work:** A unit of work in which the queue manager acts as the unit-of-work coordinator, both for MQ resources *and* for resources belonging to other resource managers. Those resource managers cooperate with the queue manager to ensure that all changes to resources in the unit of work are committed or backed out together.
 - To start this type of unit of work, use the MQBEGIN call.
 - To commit or back out this type of unit of work, use the MQCMIT and MQBACK calls.
 - **Externally-coordinated global unit of work:** A unit of work in which the queue manager is a participant, but the queue manager does not act as the unit-of-work coordinator. Instead, there is an external unit-of-work coordinator with which the queue manager cooperates.
 - To start this type of unit of work, use the relevant call provided by the external unit-of-work coordinator.
 If the MQBEGIN call is used to try to start the unit of work, the call fails with reason code MQRC_ENVIRONMENT_ERROR.
 - To commit or back out this type of unit of work, use the commit and back-out calls provided by the external unit-of-work coordinator.
 If you use the MQCMIT or MQBACK call to commit or back out the unit of work, the call fails with reason code MQRC_ENVIRONMENT_ERROR.
2. If the application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager”](#) on page 657 for further details.
3. An application can participate in only one unit of work at a time. The MQBEGIN call fails with reason code MQRC_UOW_IN_PROGRESS if there is already a unit of work in existence for the application, regardless of which type of unit of work it is.

4. The MQBEGIN call is not valid in an MQ MQI client environment. An attempt to use the call fails with reason code MQRC_ENVIRONMENT_ERROR.
5. When the queue manager is acting as the unit-of-work coordinator for global units of work, the resource managers that can participate in the unit of work are defined in the queue manager configuration file.
6. On IBM i, the three types of unit of work are supported as follows:
 - **Queue-manager-coordinated local unit of work** can be used only when a commitment definition does not exist at the job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
 - **Queue-manager-coordinated global unit of work** is not supported.
 - **Externally-coordinated global unit of work** can be used only when a commitment definition exists at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must have been issued for the job. If this has been done, the IBM i COMMIT and ROLLBACK operations apply to MQ resources as well as to resources belonging to other participating resource managers.

C invocation

```
MQBEGIN (Hconn, &BeginOptions, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;          /* Connection handle */
MQBO     BeginOptions;   /* Options that control the action of MQBEGIN */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQBEGIN' USING HCONN, BEGINOPTIONS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Options that control the action of MQBEGIN
01 BEGINOPTIONS.
   COPY CMQBOV.
** Completion code
01 COMPCODE       PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON         PIC S9(9) BINARY.
```

PL/I invocation

```
call MQBEGIN (Hconn, BeginOptions, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn          fixed bin(31); /* Connection handle */
dcl BeginOptions   like MQBO;    /* Options that control the action of
                                   MQBEGIN */
dcl CompCode       fixed bin(31); /* Completion code */
dcl Reason         fixed bin(31); /* Reason code qualifying CompCode */
```

Visual Basic invocation

```
MQBEGIN Hconn, BeginOptions, CompCode, Reason
```

Declare the parameters as follows:

Dim Hconn	As Long	'Connection handle'
Dim BeginOptions	As MQBO	'Options that control the action of MQBEGIN'
Dim CompCode	As Long	'Completion code'
Dim Reason	As Long	'Reason code qualifying CompCode'

MQBUFMH - Convert buffer into message handle

The MQBUFMH function call converts a buffer into a message handle and is the inverse of the MQMHBUF call.

This call takes a message descriptor and MQRFH2 properties in the buffer and makes them available through a message handle. The MQRFH2 properties in the message data are, optionally, removed. The *Encoding*, *CodedCharSetId*, and *Format* fields of the message descriptor are updated, if necessary, to correctly describe the contents of the buffer after the properties have been removed.

Syntax

MQBUFMH (*Hconn*, *Hmsg*, *BufMsgHOpts*, *MsgDesc*, *BufferLength*, *Buffer*, *DataLength*, *Compcode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using MQHC_UNASSOCIATED_HCONN, a valid connection must be established on the thread converting a buffer into a message handle. If a valid connection is not established, the call fails with MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMQSG - input

This is the message handle for which a buffer is required. The value was returned by a previous MQCRTMH call.

BufMsgHOpts

Type: MQBMHO - input

The MQBMHO structure allows applications to specify options that control how message handles are produced from buffers.

See [“MQBMHO - Buffer to message handle options”](#) on page 267 for details.

MsgDesc

Type: MQMD - input/output

The *MsgDesc* structure contains the message descriptor properties and describes the contents of the buffer area.

On output from the call, the properties are optionally removed from the buffer area and, in this case, the message descriptor is updated to correctly describe the buffer area.

Data in this structure must be in the character set and encoding of the application.

BufferLength

Type: MQLONG - input

BufferLength is the length of the Buffer area, in bytes.

A *BufferLength* of zero bytes is valid, and indicates that the buffer area contains no data.

Buffer

Type: MQBYTEExBufferLength - input/output

These are options that control the action of MQBEGIN, as described in [“MQBEGIN - Begin unit of work”](#) on page 604.

Buffer defines the area containing the message buffer. For most data, you should align the buffer on a 4-byte boundary.

If *Buffer* contains character or numeric data, set the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to the values appropriate to the data; this enables the data to be converted, if necessary.

If properties are found in the message buffer they are optionally removed; they later become available from the message handle on return from the call.

In the C programming language, the parameter is declared as a pointer-to-void, which means the address of any type of data can be specified as the parameter.

If the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

DataLength

Type: MQLONG - output

The length, in bytes, of the buffer which might have the properties removed.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BMHO_ERROR

(2489, X'09B9') Buffer to message handle options structure not valid.

MQRC_BUFFER_ERROR

(2004, X'07D4') Buffer parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'07D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CONNECTION_BROKEN

(2009, X'07D9') Connection to queue manager lost.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle not valid.

MQRC_MD_ERROR

(2026, X'07EA') Message descriptor not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_RFH_ERROR

(2334, X'091E') MQRFH2 structure not valid.

MQRC_RFH_FORMAT_ERROR

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

MQBUFMMH calls cannot be intercepted by API exits - a buffer is converted into a message handle in the application space; the call does not reach the queue manager.

C invocation

```
MQBUFMMH (Hconn, Hmsg, &BufMsgHOpts, &MsgDesc, BufferLength, Buffer,
          &DataLength, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN Hconn;          /* Connection handle */
MQHMSG Hmsg;             /* Message handle */
MQBMHO BufMsgHOpts;     /* Options that control the action of MQBUFMMH */
MQMD MsgDesc;           /* Message descriptor */
MQLONG BufferLength;     /* Length in bytes of the Buffer area */
MQBYTE Buffer[n];        /* Area to contain the message buffer */
MQLONG DataLength;      /* Length of the output buffer */
MQLONG CompCode;        /* Completion code */
MQLONG Reason;          /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQBUFMMH' USING HCONN, HMSG, BUFMSGHOPTS, MSGDESC, BUFFERLENGTH,
                     BUFFER, DATALENGTH, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Message handle
01 HMSG           PIC S9(18) BINARY.
** Options that control the action of MQBUFMMH
01 BUFMSGHOPTS.
   COPY CMQBMHOV.
** Message descriptor
01 MSGDESC.
   COPY CMQMD.
** Length in bytes of the Buffer area
01 BUFFERLENGTH  PIC S9(9) BINARY.
** Area to contain the message buffer
01 BUFFER        PIC X(n).
** Length of the output buffer
01 DATALENGTH   PIC S9(9) BINARY.
** Completion code
01 COMPCODE      PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON        PIC S9(9) BINARY.
```

PL/I invocation

```
call MQBUFMH (Hconn, Hmsg, BufMsgHOpts, MsgDesc, BufferLength, Buffer,
DataLength, CompCode, Reason);
```

Declare the parameters as follows:

```
dc1 Hconn          fixed bin(31); /* Connection handle */
dc1 Hmsg           fixed bin(63); /* Message handle */
dc1 BufMsgHOpts    like MQBMHO;  /* Options that control the action of
                                MQBUFMH */
dc1 MsgDesc        like MQMD;    /* Message descriptor */
dc1 BufferLength    fixed bin(31); /* Length in bytes of the Buffer area */
dc1 Buffer          char(n);      /* Area to contain the message buffer */
dc1 DataLength     fixed bin(31); /* Length of the output buffer */
dc1 CompCode       fixed bin(31); /* Completion code */
dc1 Reason         fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQBUFMH, (HCONN,HMSG,BUFMSGHOPTS,MSGDESC,BUFFERLENGTH,BUFFER,
DATALENGTH,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HMSG	DS	D	Message handle
BUFMSGHOPTS	CMQBMHOA	,	Options that control the action of MQBUFMH
MSGDESC	CMQMDA	,	Message descriptor
BUFFERLENGTH	DS	F	Length in bytes of the BUFFER area
BUFFER	DS	CL(n)	Area to contain the properties
DATALENGTH	DS	F	Length of the output buffer
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQCB - Manage callback

The MQCB call registers a callback for the specified object handle and controls activation and changes to the callback.

A callback is a piece of code (specified as either the name of a function that can be dynamically linked or as function pointer) that is called by IBM MQ when certain events occur.

To use MQCB and MQCTL on a client you must be connected to a server where the negotiated **SHARECNV** parameter of the channel has agreed a non-zero value.

The types of callback that can be defined are:

Message consumer

A message consumer callback function is called when a message, meeting the selection criteria specified, is available on an object handle.

Only one callback function can be registered against each object handle. If a single queue is to be read with multiple selection criteria then the queue must be opened multiple times and a consumer function registered on each handle.

Event handler

The event handler is called for conditions that affect the whole callback environment.

The function is called when an event condition occurs, for example, a queue manager or connection stopping or quiescing.

The function is not called for conditions that are specific to a single message consumer, for example MQRC_GET_INHIBITED; it is called however if a callback function does not end normally.

Syntax

MQCB (*Hconn*, *Operation*, *CallbackDesc*, *Hobj*, *MsgDesc*, *GetMsgOpts*, *CompCode*, *Reason*)


Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, you can specify the following special value for *MQHC_DEF_HCONN* to use the connection handle associated with this execution unit.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Operation

Type: MQLONG - input

The operation being processed on the callback defined for the specified object handle. You must specify one of the following options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

MQOP_REGISTER

Define the callback function for the specified object handle. This operation defines the function to be called and the selection criteria to be used.

If a callback function is already defined for the object handle the definition is replaced. If an error is detected while replacing the callback, the function is deregistered.

If a callback is registered in the same callback function in which it was previously deregistered, this is treated as a replace operation; any initial or final calls are not invoked.

You can use MQOP_REGISTER with MQOP_SUSPEND or MQOP_RESUME.

MQOP_DEREGISTER

Stop the consuming of messages for the object handle and removes the handle from those eligible for a callback.

A callback is automatically deregistered if the associated handle is closed.

If MQOP_DEREGISTER is called from within a consumer, and the callback has a stop call defined, it is invoked upon return from the consumer.

If this operation is issued against an *Hobj* with no registered consumer, the call returns with MQRC_CALLBACK_NOT_REGISTERED.

MQOP_SUSPEND

Suspends the consuming of messages for the object handle.

If this operation is applied to an event handler, the event handler does not get events while suspended, and any events missed while in the suspended state are not provided to the operation when it is resumed.

While suspended, the consumer function continues to get the control type callbacks.

MQOP_RESUME

Resume the consuming of messages for the object handle.

If this operation is applied to an event handler, the event handler does not get events while suspended, and any events missed while in the suspended state are not provided to the operation when it is resumed.

CallbackDesc

Type: MQCBD - input

This is a structure that identifies the callback function that is being registered by the application and the options used when registering it.

See [MQCBD](#) for details of the structure.

Callback descriptor is required only for the MQOP_REGISTER option; if the descriptor is not required, the parameter address passed can be null.

Hobj

Type: MQHOBJ - input

This handle represents the access that has been established to the object from which a message is to be consumed. This is a handle that has been returned from a previous [MQOPEN](#) or [MQSUB](#) call (in the *Hobj* parameter).

Hobj is not required when defining an event handler routine (MQCBT_EVENT_HANDLER) and should be specified as MQHO_NONE.

If *Hobj* has been returned from an MQOPEN call, the queue must have been opened with one or more of the following options:

- MQOO_INPUT_SHARED
- MQOO_INPUT_EXCLUSIVE
- MQOO_INPUT_AS_Q_DEF
- MQOO_BROWSE

MsgDesc

Type: MQMD - input

This structure describes the attributes of the message required, and the attributes of the message retrieved.

The *MsgDesc* parameter defines the attributes of the messages required by the consumer, and the version of the MQMD to be passed to the message consumer.

The *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* in the MQMD are used for message selection, depending on the options specified in the *GetMsgOpts* parameter.

The *Encoding* and *CodedCharSetId* are used for message conversion if you specify the MQGMO_CONVERT option.

See [MQMD](#) for details.

MsgDesc is used for MQOP_REGISTER and if you require values other than the default for any fields. *MsgDesc* is not used for an event handler.

If the descriptor is not required the parameter address passed can be null.

Note, that if multiple consumers are registered against the same queue with overlapping selectors, the chosen consumer for each message is undefined.

GetMsgOpts

Type: MQGMO - input

The *GetMsgOpts* parameter controls how the message consumer gets messages. All options of this parameter have meanings as described in [“MQGMO - Get-message options”](#) on page 350, when used on an MQGET call, except:

MQGMO_SET_SIGNAL

This option is not permitted.

MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT, MQGMO_MARK_*

The order of messages delivered to a browsing consumer is dictated by the combinations of these options. Significant combinations are:

MQGMO_BROWSE_FIRST

The first message on the queue is delivered repeatedly to the consumer. This is useful when the consumer destructively consumes the message in the callback. Use this option with care.

MQGMO_BROWSE_NEXT

The consumer is given each message on the queue, from the current cursor position until the end of the queue is reached.

MQGMO_BROWSE_FIRST + MQGMO_BROWSE_NEXT

The cursor is reset to the start of the queue. The consumer is then given each message until the cursor reaches the end of the queue.

MQGMO_BROWSE_FIRST + MQGMO_MARK_*

Starting at the beginning of the queue, the consumer is given the first nonmarked message on the queue, which is then marked for this consumer. This combination ensures that the consumer can receive new messages added behind the current cursor point.

MQGMO_BROWSE_NEXT + MQGMO_MARK_*

Starting at the cursor position, the consumer is given the next nonmarked message on the queue, which is then marked for this consumer. Use this combination with care because messages can be added to the queue behind the current cursor position.

MQGMO_BROWSE_FIRST + MQGMO_BROWSE_NEXT + MQGMO_MARK_*

This combination is not permitted. If used the call returns MQRC_OPTIONS_ERROR.

MQGMO_NO_WAIT, MQGMO_WAIT, and WaitInterval

These options control how the consumer is invoked.

MQGMO_NO_WAIT

The consumer is never called with MQRC_NO_MSG_AVAILABLE. The consumer is only called for messages and events.

MQGMO_WAIT with a zero WaitInterval

The MQRC_NO_MSG_AVAILABLE code is passed to the consumer when there are no messages available and either the consumer has been started or the consumer has been delivered at least one message since the last "no messages" reason code.

This prevents the consumer from polling in a busy loop when a zero wait interval is specified.

MQGMO_WAIT and a positive WaitInterval

The consumer is called after the specified wait interval with reason code MQRC_NO_MSG_AVAILABLE. This call is made regardless of whether any messages have been delivered to the consumer. This allows the user to perform heartbeat or batch type processing.

MQGMO_WAIT and WaitInterval of MQWI_UNLIMITED

This specifies an infinite wait before returning MQRC_NO_MSG_AVAILABLE. The consumer is never called with MQRC_NO_MSG_AVAILABLE.

GetMsgOpts is used only for MQOP_REGISTER and if you require values other than the default for any fields. *GetMsgOpts* is not used for an event handler.

If the *GetMsgOpts* are not required, the parameter address passed can be null. Using this parameter is the same as specifying MQGMO_DEFAULT together with MQGMO_FAIL_IF_QUIESCING.

If a message properties handle is provided in the MQGMO structure, a copy is provided in the MQGMO structure that is passed into the consumer callback. On return from the MQCB call, the application can delete the message properties handle.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason codes in the following list are the ones that the queue manager can return for the *Reason* parameter.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_CONV_LOAD_ERROR

(2133, X'855') Unable to load data conversion services modules.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CALLBACK_LINK_ERROR

(2487, X'9B7') Incorrect callback type field.

MQRC_CALLBACK_NOT_REGISTERED

(2448, X'990') Unable to unregister, suspend, or resume because there is no registered callback.

MQRC_CALLBACK_ROUTINE_ERROR

(2486, X'9B6') Either *CallbackFunction* or *CallbackName* must be specified but not both.

MQRC_CALLBACK_TYPE_ERROR

(2483, X'9B3') Incorrect callback type field.

MQRC_CBD_OPTIONS_ERROR

(2484, X'9B4') Incorrect MQCBD options field.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CORREL_ID_ERROR

(2207, X'89F') Correlation-identifier error.

MQRC_DATA_LENGTH_ERROR

(2010, X'7DA') Data length parameter not valid.

MQRC_FUNCTION_NOT_SUPPORTED

(2298, X'8FA') The function requested is not available in the current environment.

MQRC_GET_INHIBITED

(2016, X'7E0') Gets inhibited for the queue.

MQRC_GLOBAL_UOW_CONFLICT

(2351, X'92F') Global units of work conflict.

MQRC_GMO_ERROR

(2186, X'88A') Get-message options structure not valid.

MQRC_HANDLE_IN_USE_FOR_UOW

(2353, X'931') Handle in use for global unit of work.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HOBJ_ERROR

(2019, X'7E3') Object handle not valid.

MQRC_INCONSISTENT_BROWSE

(2259, X'8D3') Inconsistent browse specification.

MQRC_INCONSISTENT_UOW

(2245, X'8C5') Inconsistent unit-of-work specification.

MQRC_INVALID_MSG_UNDER_CURSOR

(2246, X'8C6') Message under cursor not valid for retrieval.

MQRC_LOCAL_UOW_CONFLICT

(2352, X'930') Global unit of work conflicts with local unit of work.

MQRC_MATCH_OPTIONS_ERROR

(2247, X'8C7') Match options not valid.

MQRC_MAX_MSG_LENGTH_ERROR

(2485, X'9B4') Incorrect *MaxMsgLength* field.

MQRC_MD_ERROR

(2026, X'7EA') Message descriptor not valid.

MQRC_MODULE_ENTRY_NOT_FOUND

(2497, X'9C1') The specified function entry point could not be found in the module.

MQRC_MODULE_INVALID

(2496, X'9C0') Module found, however it is of the wrong type; not 32 bit, 64 bit, or a valid dynamic link library.

MQRC_MODULE_NOT_FOUND

(2495, X'9BF') Module not found in the search path or not authorized to load.

MQRC_MSG_SEQ_NUMBER_ERROR

(2250, X'8CA') Message sequence number not valid.

MQRC_MSG_TOKEN_ERROR

(2331, X'91B') Use of message token not valid.

MQRC_NO_MSG_AVAILABLE

(2033, X'7F1') No message available.

MQRC_NO_MSG_UNDER_CURSOR

(2034, X'7F2') Browse cursor not positioned on message.

MQRC_NOT_OPEN_FOR_BROWSE

(2036, X'7F4') Queue not open for browse.

MQR_C_NOT_OPEN_FOR_INPUT
(2037, X'7F5') Queue not open for input.

MQR_C_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.

MQR_C_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQR_C_OPERATION_ERROR
(2206, X'89E') Incorrect operation code on API Call.

MQR_C_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.

MQR_C_PAGESET_ERROR
(2193, X'891') Error accessing page-set data set.

MQR_C_Q_DELETED
(2052, X'804') Queue has been deleted.

MQR_C_Q_INDEX_TYPE_ERROR
(2394, X'95A') Queue has wrong index type.

MQR_C_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.

MQR_C_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.

MQR_C_Q_MGR QUIESCING
(2161, X'871') Queue manager quiescing.

MQR_C_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.

MQR_C_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.

MQR_C_SIGNAL_OUTSTANDING
(2069, X'815') Signal outstanding for this handle.

MQR_C_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.

MQR_C_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.

MQR_C_SYNCPOINT_LIMIT_REACHED
(2024, X'7E8') No more messages can be handled within current unit of work.

MQR_C_SYNCPOINT_NOT_AVAILABLE
(2072, X'818') Sync point support not available.

MQR_C_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

MQR_C_UOW_ENLISTMENT_ERROR
(2354, X'932') Enlistment in global unit of work failed.

MQR_C_UOW_MIX_NOT_SUPPORTED
(2355, X'933') Mixture of unit-of-work calls not supported.

MQR_C_UOW_NOT_AVAILABLE
(2255, X'8CF') Unit of work not available for the queue manager to use.

MQR_C_WAIT_INTERVAL_ERROR
(2090, X'82A') Wait interval in MQGMO not valid.

MQR_C_WRONG_GMO_VERSION
(2256, X'8D0') Wrong version of MQGMO supplied.

MQR_C_WRONG_MD_VERSION
(2257, X'8D1') Wrong version of MQMD supplied.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. MQCB is used to define the action to be invoked for each message, matching the specified criteria, available on the queue. When the action is processed, either the message is removed from the queue and passed to the defined message consumer, or a message token is provided, which is used to retrieve the message.
2. MQCB can be used to define callback routines before starting consumption with MQCTL or it can be used from within a callback routine.
3. To use MQCB from outside of a callback routine, you must first suspend message consumption by using MQCTL and resume consumption afterward.
4. MQCB is not supported within the IMS adapter.

Message consumer callback sequence

You can configure a consumer to invoke callback at key points during the lifecycle of the consumer. For example:

- when the consumer is first registered,
- when the connection is started,
- when the connection is stopped and
- when the consumer is deregistered, either explicitly, or implicitly by an MQCLOSE.

Table 550. MQCTL verb definitions	
Verb	Meaning
MQCTL(START)	MQCTL call using the MQOP_START Operation
MQCTL(STOP)	MQCTL call using the MQOP_STOP Operation
MQCTL(WAIT)	MQCTL call using the MQOP_START_WAIT Operation

This allows the consumer to maintain state associated with the consumer. When a callback is requested by an application, the rules for consumer invocation are as follows:

REGISTER

Is always the first type of invocation of the callback.

Is always called on the same thread, as the MQCB(REGISTER) call.

START

Is always called synchronously with the MQCTL(START) verb.

- All START callbacks are completed before the MQCTL(START) verb returns.

Is on the same thread as the message delivery if THREAD_AFFINITY is requested.

The call with start is not guaranteed if, for example, a previous callback issues MQCTL(STOP) during the MQCTL(START).

STOP

No further messages or events are delivered after this call until the connection is restarted.

A STOP is guaranteed if the application was previously called for START, or a message, or an event.

DEREGISTER

Is always the last type of invocation of the callback.

Ensure that your application performs thread-based initialization and cleanup in the START and STOP callbacks. You can do non-thread based initialization and cleanup with REGISTER and DEREGISTER callbacks.

Do not make any assumptions about the life and availability of the thread other than what is stated. For example, do not rely on a thread staying alive beyond the last call to DEREGISTER. Similarly, when you have chosen not to use THREAD_AFFINITY, do not assume that the thread exists whenever the connection is started.

If your application has particular requirements for thread characteristics, it can always create a thread accordingly, then use MQCTL(WAIT). This has the effect of 'donating' the thread to IBM MQ for asynchronous message delivery.

Message consumer connection usage

You can configure a consumer to invoke callback at key points during the lifecycle of the consumer. For example:

- when the consumer is first registered,
- when the connection is started,
- when the connection is stopped and
- when the consumer is deregistered, either explicitly, or implicitly by an MQCLOSE.

<i>Table 551. MQCTL verb definitions</i>	
Verb	Meaning
MQCTL(START)	MQCTL call using the MQOP_START Operation
MQCTL(STOP)	MQCTL call using the MQOP_STOP Operation
MQCTL(WAIT)	MQCTL call using the MQOP_START_WAIT Operation

This allows the consumer to maintain state associated with the consumer. When a callback is requested by an application, the rules for consumer invocation are as follows:

REGISTER

Is always the first type of invocation of the callback.

Is always called on the same thread, as the MQCB(REGISTER) call.

START

Is always called synchronously with the MQCTL(START) verb.

- All START callbacks are completed before the MQCTL(START) verb returns.

Is on the same thread as the message delivery if THREAD_AFFINITY is requested.

The call with start is not guaranteed if, for example, a previous callback issues MQCTL(STOP) during the MQCTL(START).

STOP

No further messages or events are delivered after this call until the connection is restarted.

A STOP is guaranteed if the application was previously called for START, or a message, or an event.

DEREGISTER

Is always the last type of invocation of the callback.

Ensure that your application performs thread-based initialization and cleanup in the START and STOP callbacks. You can do non-thread based initialization and cleanup with REGISTER and DEREGISTER callbacks.

Do not make any assumptions about the life and availability of the thread other than what is stated. For example, do not rely on a thread staying alive beyond the last call to DEREGISTER. Similarly, when you have chosen not to use THREAD_AFFINITY, do not assume that the thread exists whenever the connection is started.

If your application has particular requirements for thread characteristics, it can always create a thread accordingly, then use MQCTL(WAIT). This has the effect of 'donating' the thread to IBM MQ for asynchronous message delivery.

C invocation

```
MQCB (Hconn, Operation, CallbackDesc, Hobj, MsgDesc,  
GetMsgOpts, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */  
MQLONG   Operation;       /* Operation being processed */  
MQCBD    CallbackDesc;    /* Callback descriptor */  
MQHOBJ   Hobj;            /* Object handle */  
MQMD     MsgDesc          /* Message descriptor attributes */  
MQGMO    GetMsgOpts       /* Message options */  
MQLONG   CompCode;        /* Completion code */  
MQLONG   Reason;          /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCB' USING HCONN, OPERATION, CBDESC, HOBJ, MSGDESC,  
GETMSGOPTS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN PIC S9(9) BINARY.  
** Operation  
01 OPERATION PIC S9(9) BINARY.  
** Callback Descriptor  
01 CBDESC.  
COPY CMQCBDV.  
01 HOBJ PIC S9(9) BINARY.  
** Message Descriptor  
01 MSGDESC.  
COPY CMQMDV.  
** Get Message Options  
01 GETMSGOPTS.  
COPY CMQGMV.  
** Completion code  
01 COMPCODE PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCB(Hconn, Operation, CallbackDesc, Hobj, MsgDesc, GetMsgOpts,  
CompCode, Reason)
```

Declare the parameters as follows:

```
dcl Hconn          fixed bin(31); /* Connection handle */  
dcl Operation      fixed bin(31); /* Operation */  
dcl CallbackDesc   like MQCBD;    /* Callback Descriptor */  
dcl Hobj           fixed bin(31); /* Object Handle */  
dcl MsgDesc        like MQMD;     /* Message Descriptor */  
dcl GetMsgOpts     like MQGMO;    /* Get Message Options */  
dcl CompCode       fixed bin(31); /* Completion code */  
dcl Reason         fixed bin(31); /* Reason code qualifying CompCode */
```

MQCB_FUNCTION - Callback function

The MQCB_FUNCTION function call is the callback function for event handling and asynchronous message consumption.

The MQCB_FUNCTION call definition is provided solely to describe the parameters that are passed to the callback function. No entry point called MQCB_FUNCTION is provided by the queue manager.

The specification of the actual function to be called is an input to the [MQCB](#) call and is passed in through the [MQCBD](#) structure.

Syntax

MQCB_FUNCTION (*Hconn*, *MsgDesc*, *GetMsgOpts*, *Buffer*, *Context*)

Parameters


Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call. On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for Hconn:

MQHC_DEF_CONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

MsgDesc

Type: MQMD - input

This structure describes the attributes of the message retrieved.

See “MQMD - Message descriptor” on page 396 for details.

The version of MQMD passed is the same version as passed on the MQCB call that defined the consumer function.

The address of the MQMD is passed as null characters if a version 4 MQGMO was used to request that a Message Handle be returned instead of an MQMD.

This is an input field to the message consumer function; it is not relevant to an event handler function.

GetMsgOpts

Type: MQGMO - input

Options used to control the actions of the message consumer. This parameter also contains additional information about the message returned.

See [MQGMO](#) for details.

The version of MQGMO passed is the latest version supported.

This is an input field to the message consumer function; it is not relevant to an event handler function.

Buffer

Type: MQBYTEExBufferLength - input

This is the area containing the message data.

If no message is available for this call, or if the message contains no message data, the address of the *Buffer* is passed as nulls.

This is an input field to the message consumer function; it is not relevant to an event handler function.

Context

Type: MQCBC - input/output

This structure provides context information to the callback functions. See [“MQCBC - Callback context” on page 271](#) for details.

Usage notes

1. Be aware that if your callback routines use services that could delay or block the thread, for example, MQGET with wait, could delay the dispatch of other callbacks.
2. A separate unit of work is not automatically established for each invocation of a callback routine, so routines can either issue a commit call, or defer committing, until a logical batch of work has been processed. When the batch of work is committed, it commits the messages for all callback functions that have been invoked since the last sync point.
3. Programs invoked by CICS LINK or CICS START retrieve parameters using CICS services through named objects known as channel containers. The container names are the same as the parameter names. For more information, see your CICS documentation.
4. Callback routines can issue an MQDISC call, but not for their own connection. For example, if a callback routine has created a connection, then it can also disconnect the connection.
5. A callback routine should not, in general, rely on being invoked from the same thread each time. If required, use the MQCTLO_THREAD_AFFINITY when the connection is started.
6. When a callback routine receives a nonzero reason code, it must take appropriate action.
7. MQCB_FUNCTION is not supported within the IMS adapter.

MQCLOSE - Close object

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN and MQSUB calls.

Syntax

MQCLOSE (*Hconn*, *Hobj*, *Options*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, you can omit the MQCONN call, and specify the following value for *Hconn*:

MQHC_DEF_HCONN

Default connection handle.

Hobj

Type: MQHOBJ - input/output

This handle represents the object that is being closed. The object can be of any type. The value of *Hobj* was returned by a previous MQOPEN call.

On successful completion of the call, the queue manager sets this parameter to a value that is not a valid handle for the environment. This value is:

MQHO_UNUSABLE_HOBJ

Unusable object handle.

On z/OS, *Hobj* is set to a value that is undefined.

Options

Type: MQLONG - input

This parameter controls how the object is closed.

Only permanent dynamic queues and subscriptions can be closed in more than one way, because they must be either retained or deleted; these are queues with the *DefinitionType* attribute that has the value MQQDT_PERMANENT_DYNAMIC (see the *DefinitionType* attribute described in “Attributes for queues” on page 801). The close options are summarized in this topic.

Durable subscriptions can either be kept or removed; these are created using the MQSUB call with the MQSO_DURABLE option.

When closing the handle to a managed destination (that is the *Hobj* parameter returned on an MQSUB call which used the MQSO_MANAGED option) the queue manager cleans up any publications that have not been retrieved when the associated subscription has also been removed. The subscription is removed using the MQCO_REMOVE_SUB option on the *Hsub* parameter returned on an MQSUB call. Note MQCO_REMOVE_SUB is the default behavior on MQCLOSE for a non-durable subscription.

When closing a handle to a non-managed destination you are responsible for cleaning up the queue where publications are sent. Close the subscription using MQCO_REMOVE_SUB first and then process messages off the queue until none remain.

You must specify one option only from the following:

Dynamic queue options: These options control how permanent dynamic queues are closed.

MQCO_DELETE

The queue is deleted if either of the following is true:

- It is a permanent dynamic queue, created by a previous MQOPEN call, and there are no messages on the queue and no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *Hobj*. In this case, all the messages on the queue are purged.

In all other cases, including the case where the *Hobj* was returned on an MQSUB call, the call fails with reason code MQRC_OPTION_NOT_VALID_FOR_TYPE, and the object is not deleted.

On z/OS, if the queue is a dynamic queue that has been logically deleted, and this is the last handle for it, the queue is physically deleted. See “Usage notes” on page 627 for further details.

MQCO_DELETE_PURGE

The queue is deleted, and any messages on it purged, if either of the following is true:

- It is a permanent dynamic queue, created by a previous MQOPEN call, and there are no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *Hobj*.

In all other cases, including the case where the *Hobj* was returned on an MQSUB call, the call fails with reason code MQRC_OPTION_NOT_VALID_FOR_TYPE, and the object is not deleted.

The table shows which close options are valid, and whether the object is retained or deleted.			
Type of object or queue	MQCO_NONE	MQCO_DELETE	MQCO_DELETE_PURGE
Object other than a queue	Retained	Not valid	Not valid
Predefined queue	Retained	Not valid	Not valid
Permanent dynamic queue	Retained	Deleted if empty and no pending updates	Messages deleted; queue deleted if no pending updates

The table shows which close options are valid, and whether the object is retained or deleted. (continued)			
Type of object or queue	MQCO_NONE	MQCO_DELETE	MQCO_DELETE_PURGE
Temporary dynamic queue (call issued by creator of queue)	Deleted	Deleted	Deleted
Temporary dynamic queue (call not issued by creator of queue)	Retained	Not valid	Not valid
Distribution list	Retained	Not valid	Not valid
Managed subscription destination	Retained	Not valid	Not valid
Distribution list (subscription has been removed)	Messages deleted; queue deleted	Not valid	Not valid

Subscription closure options: These options control whether durable subscriptions are removed when the handle is closed, and whether publications still waiting to be read by the application are cleaned up. These options are only valid for use with an object handle returned in the *Hsub* parameter of an MQSUB call.

MQCO_KEEP_SUB

The handle to the subscription is closed but the subscription made is kept. Publications continue to be sent to the destination specified in the subscription. This option is only valid if the subscription was made with the option MQSO_DURABLE.

MQCO_KEEP_SUB is the default if the subscription is durable

MQCO_REMOVE_SUB

The subscription is removed and the handle to the subscription is closed.

The *Hobj* parameter of the MQSUB call is not invalidated by closure of the *Hsub* parameter and might continue to be used for MQGET or MQCB to receive the remaining publications. When the *Hobj* parameter of the MQSUB call is also closed, if it was a managed destination any unretrieved publications are removed.

MQCO_REMOVE_SUB is the default if the subscription is non-durable.

Successful completion of MQCO_REMOVE_SUB does not mean that the action completed. To check that this call has completed, see the [DELETE SUB](#) step in [Checking that async commands for distributed networks have finished](#).

These subscription closure options are summarized in the following tables.

To close a durable subscription handle but retain the subscription, use the following subscription closure options:

Task	Subscription closure option
Keep publications on an MQOPENed handle	MQCO_KEEP_SUB
Remove publications on an MQOPENed handle	Action not allowed
Keep publications on an MQSO_MANAGED handle	MQCO_KEEP_SUB
Remove publications on an MQSO_MANAGED handle	Action not allowed

To unsubscribe, either by closing a durable subscription handle and unsubscribing it or closing a non-durable subscription handle, use the following subscription closure options:

Task	Subscription closure option
Keep publications on an MQOPENed handle	MQCO_REMOVE_SUB
Remove publications on an MQOPENed handle	Action not allowed
Keep publications on an MQSO_MANAGED handle	MQCO_REMOVE_SUB

Read ahead options: The following options control what happens to non-persistent messages which have been sent to the client before an application requested them and have not yet been consumed by the application. These messages are stored in the client read ahead buffer waiting to be requested by the application and can either be discarded or consumed from the queue before the MQCLOSE is completed.

MQCO_IMMEDIATE

The object is closed immediately and any messages which have been sent to the client before an application requested them are discarded and are not available to be consumed by any application. This is the default value.

MQCO_QUIESCE

A request to close the object is made, but if any messages which have been sent to the client before an application requested them, still reside in the client read ahead buffer, the MQCLOSE call returns with a warning of MQRC_READ_AHEAD_MSGS and the object handle remains valid.

The application can then continue to use the object handle to retrieve messages until no more are available, and then close the object again. No more messages are sent to the client ahead of an application requesting them, read ahead is now turned off.

Applications are advised to use MQCO_QUIESCE rather than trying to reach a point where there are no more messages in the client read ahead buffer, because a message could arrive between the last MQGET call and the following MQCLOSE which would be discarded if MQCO_IMMEDIATE was used.

If an MQCLOSE with MQCO_QUIESCE is issued from within an asynchronous callback function, the same behavior of reading ahead messages applies. If the warning MQRC_READ_AHEAD_MSGS is returned, then the callback function is called at least one more time. When the last remaining message that was read ahead has been passed to the callback function the MQCBC ConsumerFlags field is set to MQCBCF_READA_BUFFER_EMPTY.

Default option: If you require none of the options described previously, you can use the following option:

MQCO_NONE

No optional close processing required.

This *must* be specified for:

- Objects other than queues
- Predefined queues
- Temporary dynamic queues (but only in those cases where *Hobj* is *not* the handle returned by the MQOPEN call that created the queue).
- Distribution lists

In all the above cases, the object is retained and not deleted.

If this option is specified for a temporary dynamic queue:

- The queue is deleted, if it was created by the MQOPEN call that returned *Hobj* ; any messages that are on the queue are purged.
- In all other cases the queue (and any messages on it) are retained.

If this option is specified for a permanent dynamic queue, the queue is retained and not deleted.

On z/OS, if the queue is a dynamic queue that has been logically deleted, and this is the last handle for it, the queue is physically deleted. See [“Usage notes” on page 627](#) for further details.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason codes listed are the ones that the queue manager can return for the *Reason* parameter.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_INCOMPLETE_GROUP

(2241, X'8C1') Message group not complete.

MQRC_INCOMPLETE_MSG

(2242, X'8C2') Logical message not complete.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') Coupling facility not available.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_DB2_NOT_AVAILABLE

(2342, X'926') Db2 subsystem not available.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HOBJ_ERROR

(2019, X'7E3') Object handle not valid.

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged.

MQRC_OPTION_NOT_VALID_FOR_TYPE

(2045, X'7FD') On an MQOPEN or MQCLOSE call: option not valid for object type.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_PAGESET_ERROR

(2193, X'891') Error accessing page-set data set.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_Q_NOT_EMPTY

(2055, X'807') Queue contains one or more messages or uncommitted put or get requests.

MQRC_READ_AHEAD_MSGS

(nnnn, X'xxx') The client has read ahead messages that have not yet been consumed by the application.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SECURITY_ERROR

(2063, X'80F') Security error occurred.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. When an application issues the MQDISC call, or ends either normally or abnormally, any objects that were opened by the application and are still open are closed automatically with the MQCO_NONE option.
2. The following points apply if the object being closed is a *queue*:

- If operations on the queue are performed as part of a unit of work, the queue can be closed before or after the sync point occurs without affecting the outcome of the sync point. If the queue is triggered, performing a rollback before closing the queue can cause a trigger message to be issued. For more information about trigger messages, see [Properties of trigger messages](#).
 - If the queue was opened with the MQOO_BROWSE option, the browse cursor is destroyed. If the queue is then reopened with the MQOO_BROWSE option, a new browse cursor is created (see [MQOO_BROWSE](#)).
 - If a message is currently locked for this handle at the time of the MQCLOSE call, the lock is released (see [MQGMO_LOCK](#)).
 - On z/OS, if there is an MQGET request with the MQGMO_SET_SIGNAL option outstanding against the queue handle being closed, the request is canceled (see [MQGMO_SET_SIGNAL](#)). Signal requests for the same queue but lodged against different handles (*Hobj*) are not affected (unless a dynamic queue is being deleted, in which case they are also canceled).
3. The following points apply if the object being closed is a *dynamic queue* (either permanent or temporary):
- For a dynamic queue, you can specify the MQCO_DELETE and MQCO_DELETE_PURGE options regardless of the options specified on the corresponding MQOPEN call.
 - When a dynamic queue is deleted, all MQGET calls with the MQGMO_WAIT option that are outstanding against the queue are canceled and reason code MQRC_Q_DELETED is returned. See [MQGMO_WAIT](#).

Although applications cannot access a deleted queue, the queue is not removed from the system, and associated resources are not freed, until all handles that reference the queue have been closed, and all units of work that affect the queue have been either committed or backed out.

On z/OS, a queue that has been logically deleted but not yet removed from the system prevents the creation of a new queue with the same name as the deleted queue; the MQOPEN call fails with reason code MQRC_NAME_IN_USE in this case. Also, such a queue can still be displayed using MQSC commands, even though it cannot be accessed by applications.

- When a permanent dynamic queue is deleted, if the *Hobj* handle specified on the MQCLOSE call is *not* the one that was returned by the MQOPEN call that created the queue, a check is made that the user identifier that was used to validate the MQOPEN call is authorized to delete the queue. If the MQOO_ALTERNATE_USER_AUTHORITY option was specified on the MQOPEN call, the user identifier checked is the *AlternateUserId*.

This check is not performed if:

- The handle specified is the one returned by the MQOPEN call that created the queue.
 - The queue being deleted is a temporary dynamic queue.
- When a temporary dynamic queue is closed, if the *Hobj* handle specified on the MQCLOSE call is the one that was returned by the MQOPEN call that created the queue, the queue is deleted. This occurs regardless of the close options specified on the MQCLOSE call. If there are messages on the queue, they are discarded; no report messages are generated.

If there are uncommitted units of work that affect the queue, the queue and its messages are still deleted, but the units of work do not fail. However, as described previously, the resources associated with the units of work are not freed until each of the units of work has been either committed or backed out.

4. The following points apply if the object being closed is a *distribution list*:


- The only valid close option for a distribution list is MQCO_NONE; the call fails with reason code MQRC_OPTIONS_ERROR or MQRC_OPTION_NOT_VALID_FOR_TYPE if any other options are specified.
- When a distribution list is closed, individual completion codes and reason codes are not returned for the queues in the list; only the *CompCode* and *Reason* parameters of the call are available for diagnostic purposes.

If a failure occurs closing one of the queues, the queue manager continues processing and attempts to close the remaining queues in the distribution list. The *CompCode* and *Reason* parameters of the call are set to return information describing the failure. It is possible for the completion code to be MQCC_FAILED, even though most of the queues were closed successfully. The queue that encountered the error is not identified.

If there is a failure on more than one queue, it is not defined which failure is reported in the *CompCode* and *Reason* parameters.

5. On IBM i, if the application was connected implicitly when the first MQOPEN call was issued, an implicit MQDISC occurs when the last MQCLOSE is issued.

Only applications running in compatibility mode can be connected implicitly; other applications must issue the MQCONN or MQCONNEX call to connect to the queue manager explicitly.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

C invocation

```
MQCLOSE (Hconn, &Hobj, Options, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;      /* Connection handle */
MQHOBJ   Hobj;       /* Object handle */
MQLONG   Options;    /* Options that control the action of MQCLOSE */
MQLONG   CompCode;   /* Completion code */
MQLONG   Reason;     /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCLOSE' USING HCONN, HOBJ, OPTIONS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Object handle
01 HOBJ       PIC S9(9) BINARY.
** Options that control the action of MQCLOSE
01 OPTIONS    PIC S9(9) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCLOSE (Hconn, Hobj, Options, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl Hobj       fixed bin(31); /* Object handle */
dcl Options    fixed bin(31); /* Options that control the action of
                               MQCLOSE */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQCLOSE, (HCONN,HOBJ,OPTIONS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN      DS  F  Connection handle
HOBJ       DS  F  Object handle
OPTIONS    DS  F  Options that control the action of MQCLOSE
COMPCODE   DS  F  Completion code
REASON     DS  F  Reason code qualifying COMPCODE
```

Visual Basic invocation

```
MQCLOSE Hconn, Hobj, Options, CompCode, Reason
```

Declare the parameters as follows:

```
Dim Hconn As Long 'Connection handle'
Dim Hobj As Long 'Object handle'
Dim Options As Long 'Options that control the action of MQCLOSE'
Dim CompCode As Long 'Completion code'
Dim Reason As Long 'Reason code qualifying CompCode'
```

MQCMIT - Commit changes

The MQCMIT call indicates to the queue manager that the application has reached a sync point, and that all the message gets and puts that have occurred since the last sync point are to be made permanent.

Messages put as part of a unit of work are made available to other applications; messages retrieved as part of a unit of work are deleted.

- ▶ **z/OS** On z/OS, the call is used only by batch programs (including IMS batch DL/I programs).
- ▶ **IBM i** On IBM i, this call is not supported for applications running in compatibility mode.

Important: ▶ **IBM i** If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Syntax

MQCMIT (*Hconn*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason codes listed are the ones that the queue manager can return for the *Reason* parameter.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_BACKED_OUT

(2003, X'7D3') Unit of work backed out.

MQRC_OUTCOME_PENDING

(2124, X'84C') Result of commit operation is pending.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CALL_INTERRUPTED

(2549, X'9F5') MQPUT or MQCMIT was interrupted and reconnection processing cannot reestablish a definite outcome.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged.

MQRC_OUTCOME_MIXED

(2123, X'84B') Result of commit or back-out operation is mixed.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RECONNECT_FAILED

(2548, X'9F4') After reconnecting, an error occurred reinstating the handles for a reconnectable connection.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. Use this call only when the queue manager itself coordinates the unit of work. This can be:
 - A local unit of work, where the changes affect only IBM MQ resources.
 - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting IBM MQ resources.

For further details about local and global units of work, see [“MQBEGIN - Begin unit of work”](#) on page 604.

2. In environments where the queue manager does not coordinate the unit of work, the appropriate commit call must be used instead of MQCMIT. The environment might also support an implicit commit caused by the application terminating normally.
 - On z/OS, use the following calls:
 - Batch programs (including IMS batch DL/I programs) can use the MQCMIT call if the unit of work affects only IBM MQ resources. However, if the unit of work affects both IBM MQ resources and resources belonging to other resource managers (for example, Db2), use the SRRCMIT call provided by the z/OS Recoverable Resource Service (RRS). The SRRCMIT call commits changes to resources belonging to the resource managers that have been enabled for RRS coordination.
 - CICS applications must use the EXEC CICS SYNCPOINT command to commit the unit of work explicitly. Alternatively, ending the transaction results in an implicit commit of the unit of work. The MQCMIT call cannot be used for CICS applications.
 - IMS applications (other than batch DL/I programs) must use IMS calls such as GU and CHKP to commit the unit of work. The MQCMIT call cannot be used for IMS applications (other than batch DL/I programs).
 - On IBM i, use this call for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
3. If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See [MQDISC usage notes](#) for further details.
4. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

When a unit of work is committed, the queue manager retains the group and segment information, and the application can continue putting or getting messages in the current message group or logical message.

Retaining the group and segment information when a unit of work is committed allows the application to spread a large message group or large logical message consisting of many segments across several units of work. Using several units of work is advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to restart putting or getting messages at the correct point if a system failure occurs. For details of how to restart at the correct point after a system failure, see [MQPMO_LOGICAL_ORDER](#) and [MQGMO_LOGICAL_ORDER](#).

The remaining usage notes apply only when the queue manager coordinates the units of work:

5. A unit of work has the same scope as a connection handle; all IBM MQ calls that affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *Hconn* parameter described in MQCONN for information about the scope of connection handles.

6. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
7. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but that never issues a commit or back-out call, can fill queues with messages that are not available to other applications. To guard against this, the administrator must set the *MaxUncommittedMsgs* queue-manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.
8. On UNIX and Windows systems, if the *Reason* parameter is MQRC_CONNECTION_BROKEN (with a *CompCode* of MQCC_FAILED), or MQRC_UNEXPECTED_ERROR it is possible that the unit of work was successfully committed.

C invocation

```
MQCMIT (Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;      /* Connection handle */
MQLONG   CompCode;   /* Completion code */
MQLONG   Reason;     /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCMIT' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCMIT (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQCMIT,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN      DS  F  Connection handle
COMPCODE   DS  F  Completion code
REASON     DS  F  Reason code qualifying COMPCODE
```

Visual Basic invocation

```
MQCMIT Hconn, CompCode, Reason
```

Declare the parameters as follows:


```
Dim Hconn      As Long 'Connection handle'
Dim CompCode   As Long 'Completion code'
Dim Reason     As Long 'Reason code qualifying CompCode'
```

MQCONN - Connect queue manager

The MQCONN call connects an application program to a queue manager.

It provides a queue manager connection handle, which the application uses on subsequent message queuing calls.

- On z/OS, CICS applications do not have to issue this call. These applications are connected automatically to the queue manager to which the CICS system is connected. However, the MQCONN and MQDISC calls are still accepted from CICS applications.
- On IBM i, applications running in compatibility mode do not have to issue this call. These applications are connected automatically to the queue manager when they issue the first MQOPEN call. However, the MQCONN and MQDISC calls are still accepted from IBM i applications.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Other applications (that is, applications not running in compatibility mode) must use the MQCONN or MQCONNEX call to connect to the queue manager, and the MQDISC call to disconnect from the queue manager. This is the recommended style of programming.

A client connection cannot be made on a server only installation, and a local connection cannot be made on a client only installation.

Syntax

MQCONN (*QMGrName*, *Hconn*, *CompCode*, *Reason*)

Parameters

QMGrName

Type: MQCHAR48 - input

This is the name of the queue manager to which the application wants to connect. The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but can contain trailing blanks. A null character can be used to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply in the environments indicated:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- On z/OS, names that begin or end with an underscore cannot be processed by the operations and control panels. For this reason, avoid such names.
- On IBM i, enclose names containing lowercase characters, forward slash, or percent in quotation marks when specified on commands. Do not specify these quotation marks in the *QMGrName* parameter.

If the name consists entirely of blanks, the name of the *default* queue manager is used. However, note the use of blank queue manager names described in the section on IBM MQ MQI client applications.

The name specified for *QMGrName* must be the name of a *connectable* queue manager or, if queue manager groups are being used, the name of the queue manager group..

On z/OS, the queue managers to which it is possible to connect are determined by the environment:

- For CICS, you can use only the queue manager to which the CICS system is connected. The *QMgrName* parameter must still be specified, but its value is ignored; blank characters are a suitable option.
- For IMS, only queue managers that are listed in the subsystem definition table (CSQQDEFV), and listed in the SSM table in IMS, are connectable (see usage note [6](#)).
- For z/OS batch and TSO, only queue managers that reside on the same system as the application are connectable (see usage note [6](#)).

Queue-sharing groups: On systems where several queue managers exist and are configured to form a queue-sharing group, the name of the queue-sharing group can be specified for *QMgrName* in place of the name of a queue manager. This allows the application to connect to *any* queue manager that is available in the queue-sharing group and that is on the same z/OS image as the application. The system can also be configured so that using a blank *QMgrName* connects to the queue-sharing group instead of to the default queue manager.

If *QMgrName* specifies the name of the queue-sharing group, but there is also a queue manager with that name on the system, connection is made to the latter in preference to the former. Only if that connection fails is connection to one of the queue managers in the queue-sharing group attempted.

If the connection is successful, you can use the handle returned by the MQCONN or MQCONNEX call to access *all* the resources (both shared and nonshared) that belong to the queue manager to which connection has been made. Access to these resources is subject to the typical authorization controls.

If the application issues two MQCONN or MQCONNEX calls to establish concurrent connections, and one or both calls specifies the name of the queue-sharing group, the second call returns completion code MQCC_WARNING and reason code MQRC_ALREADY_CONNECTED when it connects to the same queue manager as the first call.

Queue-sharing groups are supported only on z/OS. Connection to a queue-sharing group is supported only in the batch, RRS batch, CICS, and TSO environments. For CICS, you can use only the queue sharing group to which the CICS system is connected. You must still specify the **QMgrName** parameter, but its value is ignored; blank characters are a suitable option.



Attention: IMS is unable to connect to a queue sharing group.

IBM MQ MQI client applications: For IBM MQ MQI client applications, a connection is attempted for each client-connection channel definition with the specified queue-manager name, until one is successful. The queue manager, however, must have the same name as the specified name. If an all-blank name is specified, each client-connection channel with an all-blank queue-manager name is tried until one is successful; in this case there is no check against the actual name of the queue manager.

IBM MQ client applications are not supported in z/OS, but z/OS can act as an IBM MQ server, to which IBM MQ client applications can connect.

IBM MQ MQI client queue-manager groups: If the specified name starts with an asterisk (*), the queue manager to which connection is made might have a different name from that specified by the application. The specified name (without the asterisk) defines a *group* of queue managers that are eligible for connection. The implementation selects one from the group by trying each one in turn until one is found to which a connection can be made. The order in which connections are attempted is influenced by the client channel weight and connection affinity values of the candidate channels. If none of the queue managers in the group is available for connection, the call fails. Each queue manager is tried once only. If an asterisk alone is specified for the name, an implementation-defined default queue-manager group is used.

Queue-manager groups are supported only for applications running in an MQ-client environment; the call fails if a non-client application specifies a queue-manager name beginning with an asterisk. A group is defined by providing several client connection channel definitions with the same queue-manager name (the specified name without the asterisk), to communicate with each of the queue managers in the group. The default group is defined by providing one or more client connection

channel definitions, each with a blank queue-manager name (specifying an all-blank name therefore has the same effect as specifying a single asterisk for the name for a client application).

After connecting to one queue manager of a group, an application can specify blanks in the typical way in the queue-manager name fields in the message and object descriptors to mean the name of the queue manager to which the application has connected (the *local queue manager*). If the application needs to know this name, use the MQINQ call to inquire the *QMgrName* queue-manager attribute.

Prefixing an asterisk to the connection name implies that the application does not depend on connecting to a particular queue manager in the group. Suitable applications are:

- Applications that put messages but do not get messages.
- Applications that put request messages and then get the reply messages from a *temporary dynamic* queue.

Unsuitable applications are ones that need to get messages from a particular queue at a particular queue manager; such applications must not prefix the name with an asterisk.

If you specify an asterisk, the maximum length of the remainder of the name is 47 characters.

The length of this parameter is given by MQ_Q_MGR_NAME_LENGTH.

Hconn

Type: MQHCONN - output

This handle represents the connection to the queue manager. Specify it on all subsequent message queuing calls issued by the application. It ceases to be valid when the MQDISC call is issued, or when the unit of processing that defines the scope of the handle terminates.

IBM MQ now supplies the mqm library with client packages as well as server packages. This means that when an MQI call that is found in the mqm library is made, the connection type is checked to see if it is a client or server connection, and then the correct underlying call is made. Therefore an exit which is passed an *Hconn* can now be linked against the mqm library, but used on a client installation.

Handle scope: The scope of the handle returned depends on the call used to connect to the queue manager (MQCONN or MQCONNX). If the call used is MQCONNX, the scope of the handle also depends on the MQCNO_HANDLE_SHARE_* option specified in the *Options* field of the MQCNO structure.

- If the call is MQCONN, or the MQCNO_HANDLE_SHARE_NONE option is specified, the handle returned is a *nonshared* handle.

The scope of a nonshared handle is the smallest unit of parallel processing supported by the platform on which the application is running (see [Table 552 on page 637](#) for details); the handle is not valid outside the unit of parallel processing from which the call was issued.

- If you specify the MQCNO_HANDLE_SHARE_BLOCK or MQCNO_HANDLE_SHARE_NO_BLOCK option, the handle returned is a *shared* handle.

The scope of a shared handle is the process that owns the thread from which the call was issued; the handle can be used from any thread that belongs to that process. Not all platforms support threads.

- If the MQCONN or MQCONNX call fails with completion code equal to MQCC_FAILED, then the Hconn value is undefined.

Table 552. Scope of nonshared handles on various platforms	
Platform	Scope of nonshared handle
z/OS	<ul style="list-style-type: none"> • CICS: the CICS task • IMS: the task, up to the next sync point (excluding subtasks of the task) • z/OS batch and TSO: the task (excluding subtasks of the task)
IBM i	Job
UNIX systems	Thread
16 bit Windows applications	Process
32 bit Windows applications	Thread

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the value returned is:

MQHC_DEF_HCONN

Default connection handle.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_ALREADY_CONNECTED

(2002, X'7D2') Application already connected.

MQRC_CLUSTER_EXIT_LOAD_ERROR

(2267, X'8DB') Unable to load cluster workload exit.

MQRC_SSL_ALREADY_INITIALIZED

(2391, X'957') SSL already initialized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_CONN_LOAD_ERROR

(2129, X'851') Unable to load adapter connection module.

MQRC_ADAPTER_DEFS_ERROR

(2131, X'853') Adapter subsystem definition module not valid.

MQRC_ADAPTER_DEFS_LOAD_ERROR

(2132, X'854') Unable to load adapter subsystem definition module.

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ADAPTER_STORAGE_SHORTAGE

(2127, X'84F') Insufficient storage for adapter.

MQRC_ANOTHER_Q_MGR_CONNECTED

(2103, X'837') Another queue manager already connected.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_INIT_ERROR

(2375, X'947') API exit initialization failed.

MQRC_API_EXIT_TERM_ERROR

(2376, X'948') API exit termination failed.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CONN_ID_IN_USE

(2160, X'870') Connection identifier already in use.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_ERROR

(2273, X'8E1') Error processing MQCONN call.

MQRC_CONNECTION_NOT_AVAILABLE

(2568, X'A08') Occurs on an MQCONN or MQCONNEX call when the queue manager is unable to provide a connection of the requested connection type on the current installation. A client connection cannot be made on a server only installation. A local connection cannot be made on a client only installation.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CRYPTO_HARDWARE_ERROR

(2382, X'94E') Cryptographic hardware configuration error.

MQRC_DUPLICATE_RECOV_COORD

(2163, X'873') Recovery coordinator exists.

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment.

Additionally, on the MQCONNEX call, passing the [“MQCSP - Security parameters” on page 320](#) control block from a CICS or IMS application.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HOST_NOT_AVAILABLE

(2538, X'9EA') An MQCONN call was issued from a client to connect to a queue manager but the attempt to allocate a conversation to the remote system failed.

MQRC_INSTALLATION_MISMATCH

(2583, X'A17') Mismatch between queue manager installation and selected library.

MQRC_KEY_REPOSITORY_ERROR

(2381, X'94D') Key repository not valid.

MQRC_MAX_CONNS_LIMIT_REACHED

(2025, X'7E9') Maximum number of connections reached.

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_OPEN_FAILED

(2137, X'859') Object not opened successfully.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SECURITY_ERROR

(2063, X'80F') Security error occurred.

MQRC_SSL_INITIALIZATION_ERROR

(2393, X'959') SSL initialization error.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. The queue manager to which connection is made using the MQCONN call is called the *local queue manager*.
2. Queues that are owned by the local queue manager appear to the application as local queues. It is possible to put messages on and get messages from these queues.

Shared queues that are owned by the queue-sharing group to which the local queue manager belongs appear to the application as local queues. It is possible to put messages on and get messages from these queues.

Queues that are owned by remote queue managers appear as remote queues. It is possible to put messages on these queues, but not to get messages from these queues.
3. If the queue manager fails while an application is running, the application must issue the MQCONN call again to obtain a new connection handle to use on subsequent IBM MQ calls. The application can issue the MQCONN call periodically until the call succeeds.

If an application is not sure whether it is connected to the queue manager, the application can safely issue an MQCONN call to obtain a connection handle. If the application is already connected, the handle returned is the same as that returned by the previous MQCONN call, but with completion code MQCC_WARNING and reason code MQRC_ALREADY_CONNECTED.
4. When the application has finished using IBM MQ calls, the application must use the MQDISC call to disconnect from the queue manager.
5. If the MQCONN call fails with completion code equal to MQCC_FAILED, then the Hconn value is undefined.

6. On z/OS:

- Batch, TSO, and IMS applications must issue the MQCONN call to use the other IBM MQ calls. These applications can connect to more than one queue manager concurrently.

If the queue manager fails, the application must issue the call again after the queue manager has restarted to obtain a new connection handle.

Although IMS applications can issue the MQCONN call repeatedly, even when already connected, this is not recommended for online message processing programs (MPPs).

- CICS applications do not have to issue the MQCONN call to use the other IBM MQ calls, but can do so if they want; both the MQCONN call and the MQDISC call are accepted. However, it is not possible to connect to more than one queue manager concurrently.

If the queue manager fails, these applications are automatically reconnected when the queue manager restarts, and so do not need to issue the MQCONN call.

7. On z/OS, to define the available queue managers:

- For batch applications, system programmers can use the CSQBDEF macro to create a module (CSQBDEFV) that defines the default queue-manager name, or queue-sharing group name.
- For IMS applications, system programmers can use the CSQQDEFX macro to create a module (CSQQDEFV) that defines the names of the available queue managers and specifies the default queue manager.

In addition, each queue manager must be defined to the IMS control region and to each dependent region accessing that queue manager. To do this, you must create a subsystem member in the IMS.PROCLIB library and identify the subsystem member to the applicable IMS regions. If an application attempts to connect to a queue manager that is not defined in the subsystem member for its IMS region, the application abends.

 For more information about using these macros, see [Macros intended for customer use](#).

8. On IBM i, applications written for previous releases of the queue manager can run without recompiling. This is called *compatibility mode*. This mode of operation provides a compatible runtime environment for applications. It comprises the following:

- The service program AMQZSTUB residing in the library QMQM.

AMQZSTUB provides the same public interface as previous releases, and has the same signature. Use this service program to access the MQI through bound procedure calls.

- The program QMQM residing in the library QMQM.

QMQM provides a means of accessing the MQI through dynamic program calls.

- Programs MQCLOSE, MQCONN, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET residing in the library QMQM.

These programs also provide a means of accessing the MQI through dynamic program calls, but with a parameter list that corresponds to the standard descriptions of the IBM MQ calls.

These three interfaces do not include capabilities that were introduced in IBM MQ Version 5.1. For example, the MQBACK, MQCMIT, and MQCONNX calls are not supported. The support provided by these interfaces is for single-threaded applications only.

Support for the new IBM MQ calls in single-threaded applications, and for all IBM MQ calls in multi-threaded applications, is provided through the service programs LIBMQM and LIBMQM_R.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

9. On IBM i, programs that end abnormally are not automatically disconnected from the queue manager. Write applications to allow for the possibility of the MQCONN or MQCONNX call returning completion code MQCC_WARNING and reason code MQRC_ALREADY_CONNECTED. Use the connection handle returned in this situation as normal.

C invocation

```
MQCONN (QMgrName, &Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQCHAR48  QMgrName;    /* Name of queue manager */
MQHCONN   Hconn;        /* Connection handle */
MQLONG    CompCode;     /* Completion code */
MQLONG    Reason;       /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCONN' USING QMGRNAME, HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Name of queue manager
01 QMGRNAME PIC X(48).
** Connection handle
01 HCONN PIC S9(9) BINARY.
** Completion code
01 COMPCODE PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCONN (QMgrName, Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl QMgrName char(48);    /* Name of queue manager */
dcl Hconn fixed bin(31);  /* Connection handle */
dcl CompCode fixed bin(31); /* Completion code */
dcl Reason fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQCONN,(QMGRNAME,HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
QMGRNAME DS CL48 Name of queue manager
HCONN DS F Connection handle
COMPCODE DS F Completion code
REASON DS F Reason code qualifying COMPCODE
```

Visual Basic invocation

```
MQCONN QMgrName, Hconn, CompCode, Reason
```

Declare the parameters as follows:

```
Dim QMgrName As String*48 'Name of queue manager'
Dim Hconn As Long 'Connection handle'
Dim CompCode As Long 'Completion code'
Dim Reason As Long 'Reason code qualifying CompCode'
```

MQCONN - Connect queue manager (extended)

The MQCONN call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent IBM MQ calls.

The MQCONN call is like the MQCONN call, except that MQCONN allows options to be specified to control the way that the call works.

- This call is supported on all IBM MQ systems, and IBM MQ clients connected to these systems.
- On IBM i, this call is not supported for applications running in compatibility mode.

A client connection cannot be made on a server only installation, and a local connection cannot be made on a client only installation.

Syntax

MQCONN (*QMgrName*, *ConnectOpts*, *Hconn*, *CompCode*, *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

See the *QMgrName* parameter described in [“MQCONN - Connect queue manager” on page 634](#) for details.

ConnectOpts

Type: MQCNO - input/output

See [“MQCNO - Connect options” on page 306](#) for details.

Hconn

Type: MQHCONN - output

This handle represents the connection to the queue manager. Specify it on all subsequent message queuing calls issued by the application. It ceases to be valid when the MQDISC call is issued, or when the unit of processing that defines the scope of the handle terminates.

IBM MQ now supplies the mqm library with client packages as well as server packages. This means that when an MQI call that is found in the mqm library is made, the connection type is checked to see if it is a client or server connection, and then the correct underlying call is made. Therefore an exit which is passed an *Hconn* can now be linked against the mqm library, but used on a client installation.

Handle scope: The scope of the handle returned depends on the call used to connect to the queue manager (MQCONN or MQCONN). If the call used is MQCONN, the scope of the handle also depends on the MQCNO_HANDLE_SHARE_* option specified in the *Options* field of the MQCNO structure.

- If the call is MQCONN, or the MQCNO_HANDLE_SHARE_NONE option is specified, the handle returned is a *nonshared* handle.

The scope of a nonshared handle is the smallest unit of parallel processing supported by the platform on which the application is running (see [Table 553 on page 643](#) for details); the handle is not valid outside the unit of parallel processing from which the call was issued.

- If you specify the MQCNO_HANDLE_SHARE_BLOCK or MQCNO_HANDLE_SHARE_NO_BLOCK option, the handle returned is a *shared* handle.

The scope of a shared handle is the process that owns the thread from which the call was issued; the handle can be used from any thread that belongs to that process. Not all platforms support threads.

- If the MQCONN or MQCONN call fails with completion code equal to MQCC_FAILED, then the *Hconn* value is undefined.

Table 553. Scope of nonshared handles on various platforms

Platform	Scope of nonshared handle
z/OS	<ul style="list-style-type: none"> • CICS: the CICS task • IMS: the task, up to the next sync point (excluding subtasks of the task) • z/OS batch and TSO: the task (excluding subtasks of the task)
IBM i	Job
UNIX systems	Thread
16 bit Windows applications	Process
32 bit Windows applications	Thread

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the value returned is:

MQHC_DEF_HCONN

Default connection handle.

CompCode

Type: MQLONG - output

See the *CompCode* parameter described in “MQCONN - Connect queue manager” on page 634 for details.

Reason

Type: MQLONG - output

The following codes can be returned by the MQCONN and MQCONNX calls. For a list of additional codes that can be returned by the MQCONNX call, see the following codes.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_ALREADY_CONNECTED

(2002, X'7D2') Application already connected.

MQRC_CLUSTER_EXIT_LOAD_ERROR

(2267, X'8DB') Unable to load cluster workload exit.

MQRC_SSL_ALREADY_INITIALIZED

(2391, X'957') SSL already initialized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_CONN_LOAD_ERROR

(2129, X'851') Unable to load adapter connection module.

MQRC_ADAPTER_DEFS_ERROR

(2131, X'853') Adapter subsystem definition module not valid.

MQRC_ADAPTER_DEFS_LOAD_ERROR

(2132, X'854') Unable to load adapter subsystem definition module.

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ADAPTER_STORAGE_SHORTAGE

(2127, X'84F') Insufficient storage for adapter.

MQRC_ANOTHER_Q_MGR_CONNECTED

(2103, X'837') Another queue manager already connected.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_INIT_ERROR

(2375, X'947') API exit initialization failed.

MQRC_API_EXIT_TERM_ERROR

(2376, X'948') API exit termination failed.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CONN_ID_IN_USE

(2160, X'870') Connection identifier already in use.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_ERROR

(2273, X'8E1') Error processing MQCONN call.

MQRC_CONNECTION_NOT_AVAILABLE

(2568, X'A08') Occurs on an MQCONN or MQCONNEX call when the queue manager is unable to provide a connection of the requested connection type on the current installation. A client connection cannot be made on a server only installation. A local connection cannot be made on a client only installation.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CRYPTO_HARDWARE_ERROR

(2382, X'94E') Cryptographic hardware configuration error.

MQRC_DUPLICATE_RECOV_COORD

(2163, X'873') Recovery coordinator exists.

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment.

Additionally, on the MQCONNEX call, passing the [“MQCSP - Security parameters” on page 320](#) control block from a CICS or IMS application.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HOST_NOT_AVAILABLE

(2538, X'9EA') An MQCONN call was issued from a client to connect to a queue manager but the attempt to allocate a conversation to the remote system failed.

MQRC_INSTALLATION_MISMATCH

(2583, X'A17') Mismatch between queue manager installation and selected library.

MQRC_KEY_REPOSITORY_ERROR

(2381, X'94D') Key repository not valid.

MQRC_MAX_CONNS_LIMIT_REACHED

(2025, X'7E9') Maximum number of connections reached.

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_OPEN_FAILED

(2137, X'859') Object not opened successfully.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SECURITY_ERROR

(2063, X'80F') Security error occurred.

MQRC_SSL_INITIALIZATION_ERROR

(2393, X'959') SSL initialization error.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

The following additional reason codes can be returned by the MQCONN call:

If *CompCode* is MQCC_FAILED:

MQRC_AIR_ERROR

(2385, X'951') Authentication information record not valid.

MQRC_AUTH_INFO_CONN_NAME_ERROR

(2387, X'953') Authentication information connection name not valid.

MQRC_AUTH_INFO_REC_COUNT_ERROR

(2383, X'94F') Authentication information record count not valid.

MQRC_AUTH_INFO_REC_ERROR

(2384, X'950') Authentication information record fields not valid.

MQRC_AUTH_INFO_TYPE_ERROR

(2386, X'952') Authentication information type not valid.

MQRC_CD_ERROR

(2277, X'8E5') Channel definition not valid.

MQRC_CLIENT_CONN_ERROR

(2278, X'8E6') Client connection fields not valid.

MQRC_CNO_ERROR

(2139, X'85B') Connect-options structure not valid.

MQRC_CONN_TAG_IN_USE

(2271, X'8DF') Connection tag in use.

MQRC_CONN_TAG_NOT_USABLE

(2350, X'92E') Connection tag not usable.

MQRC_LDAP_PASSWORD_ERROR

(2390, X'956') LDAP password not valid.

MQRC_LDAP_USER_NAME_ERROR

(2388, X'954') LDAP user name fields not valid.

MQRC_LDAP_USER_NAME_LENGTH_ERR

(2389, X'955') LDAP user name length not valid.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_SCO_ERROR

(2380, X'94C') SSL configuration options structure not valid.

MQRC_SSL_CONFIG_ERROR

(2392, X'958') SSL configuration error.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

For the Visual Basic programming language, the following point applies:

- The *ConnectOpts* parameter is declared as being of type MQCNO. If the application is running as an IBM MQ MQI client, and you want to specify the parameters of the client-connection channel, declare the *ConnectOpts* parameter as being of type Any, so that the application can specify an MQCNOCD structure on the call in place of an MQCNO structure. However, this means that the *ConnectOpts* parameter cannot be checked to ensure that it is the correct data type.

C invocation

```
MQCONN (QMgrName, &ConnectOpts, &Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQCHAR48  QMgrName;      /* Name of queue manager */
MQCNO     ConnectOpts;   /* Options that control the action of MQCONN */
MQHCONN   Hconn;         /* Connection handle */
MQLONG    CompCode;      /* Completion code */
MQLONG    Reason;        /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCONN' USING QMGRNAME, CONNECTOPTS, HCONN, COMPCODE,
REASON.
```

Declare the parameters as follows:

```
** Name of queue manager
01 QMGRNAME      PIC X(48).
** Options that control the action of MQCONN
01 CONNECTOPTS.
   COPY CMQCNV.
** Connection handle
01 HCONN        PIC S9(9) BINARY.
** Completion code
01 COMPCODE     PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON       PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCONN (QMgrName, ConnectOpts, Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl QMgrName      char(48);      /* Name of queue manager */
dcl ConnectOpts   like MQCNO;    /* Options that control the action of
                                   MQCONN */
dcl Hconn         fixed bin(31); /* Connection handle */
```

```

dcl CompCode      fixed bin(31); /* Completion code */
dcl Reason        fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```
CALL MQCONN, (QMGRNAME,CONNECTOPTS,HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

QMGRNAME	DS	CL48	Name of queue manager
CONNECTOPTS	CMQCN0A	,	Options that control the action of MQCONN
HCONN	DS	F	Connection handle
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

Visual Basic invocation

```
MQCONN QMgrName, ConnectOpts, Hconn, CompCode, Reason
```

Declare the parameters as follows:

```

Dim QMgrName As String*48 'Name of queue manager'
Dim ConnectOpts As MQCNO 'Options that control the action of'
                        'MQCONN'
Dim Hconn As Long 'Connection handle'
Dim CompCode As Long 'Completion code'
Dim Reason As Long 'Reason code qualifying CompCode'

```

MQCRTMH - Create message handle

The MQCRTMH call returns a message handle.

An application can use the MQCRTMH call on subsequent message queuing calls:

- Use the [MQSETMP](#) call to set a property of the message handle.
- Use the [MQINQMP](#) call to inquire on the value of a property of the message handle.
- Use the [MQDLTMP](#) call to delete a property of the message handle.

The message handle can be used on the MQPUT and MQPUT1 calls to associate the properties of the message handle with those of the message being put. Similarly by specifying a message handle on the MQGET call, the properties of the message being retrieved can be accessed using the message handle when the MQGET call completes.

Use [MQDLTMH](#) to delete the message handle.

Syntax

```
MQCRTMH (Hconn, CrtMsgHOpts, Hmsg, CompCode, Reason)
```

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call. If the connection to the queue manager ceases to be valid and no IBM MQ call is operating on the message handle, [MQDLTMH](#) is implicitly called to delete the message.

Alternatively, you can specify the following value:

MQHC_UNASSOCIATED_HCONN

The connection handle does not represent a connection to any particular queue manager.

When this value is used, the message handle must be deleted with an explicit call to `MQDLTMH` in order to release any storage allocated to it; IBM MQ never implicitly deletes the message handle.


There must be at least one valid connection to a queue manager established on the thread creating the message handle, otherwise the call fails with `MQRC_HCONN_ERROR`.

In an environment with multiple installations on a single system, the `MQHC_UNASSOCIATED_HCONN` value is limited to use with the first installation loaded into the process. The reason code `MQRC_HMSG_NOT_AVAILABLE` is returned if the message handle is supplied to a different installation.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the `MQCONN` call can be omitted, and you can specify the following value for *Hconn* :

MQHC_DEF_CONN

Default connection handle

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

CrtMsgHOpts

Type: `MQCMHO` - input

The options that control the action of `MQCRTMH`. See [MQCMHO](#) for details.

Hmsg

Type: `MQHMSG` - output

On output a message handle is returned that can be used to set, inquire, and delete properties of the message handle. Initially the message handle contains no properties.

A message handle also has an associated message descriptor. Initially this contains the default values. The values of the associated message descriptor fields can be set and inquired using the `MQSETMP` and `MQINQMP` calls. The `MQDLTMP` call resets a field of the message descriptor back to its default value.

If the *Hconn* parameter is specified as the value `MQHC_UNASSOCIATED_HCONN` then the returned message handle can be used on `MQGET`, `MQPUT`, or `MQPUT1` calls with any connection within the unit of processing, but can only be in use by one IBM MQ call at a time. If the handle is in use when a second IBM MQ call attempts to use the same message handle, the second IBM MQ call fails with reason code `MQRC_MSG_HANDLE_IN_USE`.

If the *Hconn* parameter is not `MQHC_UNASSOCIATED_HCONN` then the returned message handle can only be used on the specified connection.

The same *Hconn* parameter value must be used on the subsequent `MQI` calls where this message handle is used:

- `MQDLTMH`
- `MQSETMP`
- `MQINQMP`
- `MQDLTMP`
- `MQMHBUF`
- `MQBUFMH`

The returned message handle ceases to be valid when the `MQDLTMH` call is issued for the message handle, or when the unit of processing that defines the scope of the handle terminates. `MQDLTMH` is called implicitly if a specific connection is supplied when the message handle is created and the connection to the queue manager ceases to be valid, for example, if `MQDBC` is called.

CompCode

Type: `MQLONG` - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CMHO_ERROR

(2461, X'099D') Create message handle options structure not valid.

MQRC_CONNECTION_BROKEN

(2273, X'7D9') Connection to queue manager lost.

MQRC_HANDLE_NOT_AVAILABLE

(2017, X'07E1') No more handles available.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle pointer not valid.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C

```
MQCRTMH (Hconn, &CrtMsgHOpts, &Hmsg, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQCMHO   CrtMsgHOpts;     /* Options that control the action of MQCRTMH */
MQHMSG   Hmsg;            /* Message handle */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying CompCode */
```

COBOL

```
CALL 'MQCRTMH' USING HCONN, CRTMSGHOPTS, HMSG, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Options that control the action of MQCRTMH
01 CRTMSGHOPTS.
   COPY CMQCMHOV.
** Message handle
01 HMSG       PIC S9(18) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.
```

PL/I

```
call MQCRTMH (Hconn, CrtMsgHOpts, Hmsg, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl CrtMsgHOpts like MQCMHO; /* Options that control the action of MQCRTMH */
dcl Hmsg       fixed bin(63); /* Message handle */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler

```
CALL MQCRTMH,(HCONN,CRTMSGHOPTS,HMSG,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
CRTMSGHOPTS	CMQCMHOA	,	Options that control the action of MQCRTMH
HMSG	DS	D	Message handle
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQCTL - Control callbacks

The MQCTL call performs controlling actions on callbacks and the object handles opened for a connection.

Syntax

MQCTL (*Hconn*, *Operation*, *ControlOpts*, *CompCode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and you can specify the following special value for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Operation

Type: MQLONG - input

The operation being processed on the callback defined for the specified object handle. You must specify one, and one only, of the following options:

MQOP_START

Start the consuming of messages for all defined message consumer functions for the specified connection handle.

Callbacks run on a thread started by the system, which is different from any of the application threads.

This operation gives control of the provided connection handle to system. The only MQI calls which can be issued by a thread other than the consumer thread are:

- MQCTL with Operation MQOP_STOP
- MQCTL with Operation MQOP_SUSPEND
- MQDISC - Performs MQCTL with Operation MQOP_STOP before disconnection the HConn.

MQRC_HCONN_ASYNC_ACTIVE is returned if an IBM MQ API call is issued while the connection handle is started, and the call does not originate from a message consumer function.

If a message consumer stops the connection during the MQCBCT_START_CALL then the MQCTL call returns with a failure reason code of MQRC_CONNECTION_STOPPED.

This can be issued in a consumer function. For the same connection as the callback routine, its only purpose is to cancel a previously issued MQOP_STOP operation.

This option is not supported in the following environments: CICS on z/OS or if the application is bound with a nonthreaded IBM MQ library.

MQOP_START_WAIT

Start the consuming of messages for all defined message consumer functions for the specified connection handle.

Message consumers run on the same thread and control is not returned to the caller of MQCTL until:

- Released by the use of the MQCTL MQOP_STOP or MQOP_SUSPEND operations, or
- All consumer routines have been deregistered or suspended.

If all consumers are deregistered or suspended, an implicit MQOP_STOP operation is issued.

This option cannot be used from within a callback routine, either for the current connection handle or any other connection handle. If the call is attempted it returns with MQRC_ENVIRONMENT_ERROR.

If, at any time during an MQOP_START_WAIT operation there are no registered, non-suspended consumers the call fails with a reason code of MQRC_NO_CALLBACKS_ACTIVE.

If, during an MQOP_START_WAIT operation, the connection is suspended, the MQCTL call returns a warning reason code of MQRC_CONNECTION_SUSPENDED; the connection remains 'started'.

The application can choose to issue MQOP_STOP or MQOP_RESUME. In this instance, the MQOP_RESUME operation blocks.

This option is not supported in a single threaded client.

MQOP_STOP

Stop the consuming of messages, and wait for all consumers to complete their operations before this option completes. This operation releases the connection handle.

If issued from within a callback routine, this option does not take effect until the routine exits. No more message consumer routines are called after the consumer routines for messages already read have completed, and after stop calls (if requested) to callback routines have been made.

If issued outside a callback routine, control does not return to the caller until the consumer routines for messages already read have completed, and after stop calls (if requested) to callbacks have been made. The callbacks themselves, however, remain registered.

This function has no effect on read ahead messages. You must ensure that consumers run MQCLOSE(MQCO_QUIESCE), from within the callback function, to determine whether there are any further messages available to be delivered.

MQOP_SUSPEND

Pause the consuming of messages. This operation releases the connection handle.

This does not have any effect on the reading ahead of messages for the application. If you intend to stop consuming messages for a long time, consider closing the queue and reopening it when consumption continues.

If issued from within a callback routine, it does not take effect until the routine exits. No more message consumer routines will be called after the current routine exits.

If issued outside a callback, control does not return to the caller until the current consumer routine has completed and no more are called.

MQOP_RESUME

Resume the consuming of messages.

This option is normally issued from the main application thread, but it can also be used from within a callback routine to cancel an earlier suspension request issued in the same routine.

If the MQOP_RESUME is used to resume an MQOP_START_WAIT then the operation blocks.

ControlOpts

Type: MQCTLO - input

Options that control the action of MQCTL

See [MQCTLO](#) for details of the structure.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_CONV_LOAD_ERROR

(2133, X'855') Unable to load data conversion services modules.

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALLBACK_LINK_ERROR

(2487, X'9B7') Unable to call the callback routine

MQRC_CALLBACK_NOT_REGISTERED

(2448, X'990') Unable to Deregister, Suspend, or Resume because there is no registered callback

MQRC_CALLBACK_ROUTINE_ERROR

(2486, X'9B6') Either, both CallbackFunction and CallbackName have been specified on an MQOP_REGISTER call.

Or either CallbackFunction or CallbackName have been specified but does not match the currently registered callback function.

MQRC_CALLBACK_TYPE_ERROR

(2483, X'9B3') Incorrect CallBackType field.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CBD_ERROR

(2444, X'98C') Option block is incorrect.

MQRC_CBD_OPTIONS_ERROR

(2484, X'9B4') Incorrect MQCBD options field.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION_QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CORREL_ID_ERROR

(2207, X'89F') Correlation-identifier error.

MQRC_FUNCTION_NOT_SUPPORTED

(2298, X'8FA') The function requested is not available in the current environment.

MQRC_GET_INHIBITED

(2016, X'7E0') Gets inhibited for the queue.

MQRC_GLOBAL_UOW_CONFLICT

(2351, X'92F') Global units of work conflict.

MQRC_GMO_ERROR

(2186, X'88A') Get-message options structure not valid.

MQRC_HANDLE_IN_USE_FOR_UOW

(2353, X'931') Handle in use for global unit of work.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_HOBJ_ERROR

(2019, X'7E3') Object handle not valid.

MQRC_INCONSISTENT_BROWSE

(2259, X'8D3') Inconsistent browse specification.

MQRC_INCONSISTENT_UOW

(2245, X'8C5') Inconsistent unit-of-work specification.

MQRC_INVALID_MSG_UNDER_CURSOR

(2246, X'8C6') Message under cursor not valid for retrieval.

MQRC_LOCAL_UOW_CONFLICT

(2352, X'930') Global unit of work conflicts with local unit of work.

MQRC_MATCH_OPTIONS_ERROR

(2247, X'8C7') Match options not valid.

MQRC_MAX_MSG_LENGTH_ERROR

(2485, X'9B5') Incorrect MaxMsgLength field

MQRC_MD_ERROR

(2026, X'7EA') Message descriptor not valid.

MQRC_MODULE_ENTRY_NOT_FOUND

(2497, X'9C1') The specified function entry point could not be found in the module.

MQRC_MODULE_INVALID

(2496, X'9C0') Module is found but is of the wrong type (32 bit/64 bit) or is not a valid dll.

MQRC_MODULE_NOT_FOUND

(2495, X'9BF') Module not found in the search path or not authorized to load.

MQRC_MSG_ID_ERROR

(2206, X'89E') Message-identifier error.

MQRC_MSG_SEQ_NUMBER_ERROR

(2250, X'8CA') Message sequence number not valid.

MQRC_MSG_TOKEN_ERROR

(2331, X'91B') Use of message token not valid.

MQRC_NOT_OPEN_FOR_BROWSE

(2036, X'7F4') Queue not open for browse.

MQRC_NOT_OPEN_FOR_INPUT

(2037, X'7F5') Queue not open for input.

MQRC_OBJECT_CHANGED

(2041, X'7F9') Object definition changed since opened.

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged.

MQRC_OPERATION_ERROR

(2488, X'9B8') Incorrect Operation code on API Call

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_PAGESET_ERROR

(2193, X'891') Error accessing page-set data set.

MQRC_Q_DELETED

(2052, X'804') Queue has been deleted.

MQRC_Q_INDEX_TYPE_ERROR

(2394, X'95A') Queue has wrong index type.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SIGNAL_OUTSTANDING

(2069, X'815') Signal outstanding for this handle.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_SYNCPOINT_NOT_AVAILABLE

(2072, X'818') Syncpoint support not available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UOW_ENLISTMENT_ERROR

(2354, X'932') Enlistment in global unit of work failed.

MQRC_UOW_MIX_NOT_SUPPORTED

(2355, X'933') Mixture of unit-of-work calls not supported.

MQRC_UOW_NOT_AVAILABLE

(2255, X'8CF') Unit of work not available for the queue manager to use.

MQRC_WAIT_INTERVAL_ERROR

(2090, X'82A') Wait interval in MQGMO not valid.

MQRC_WRONG_GMO_VERSION

(2256, X'8D0') Wrong version of MQGMO supplied.

MQRC_WRONG_MD_VERSION

(2257, X'8D1') Wrong version of MQMD supplied.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. Callback routines must check the responses from all services they invoke, and if the routine detects a condition that cannot be resolved, it must issue an MQCB MQOP_DEREGISTER command to prevent repeated calls to the callback routine.
2. If you are using asynchronous consume in an application where an XA Transaction Manager is managing global transactions, including updates to IBM MQ, you need to consider the following additional points:
 - a. It is not valid to call MQCTL(MQOP_START) for an **HConn**, after it has been created, after calling **xa_open**.
The reason is, that the **HConn** has become attached to an XA context, and so cannot then be accessed on the separate thread, or threads, in use by the asynchronous consume mechanism.
 - b. If you call MQCTL(MQOP_START) in that scenario the call fails with reason code MQRC_ASYNC_XA_CONFLICT (2350).
 - c. It is valid to call MQCTL(MQOP_START_WAIT) for an **HConn**, after it has been created, after calling **xa_open**.

The reason is, that this method of starting the asynchronous consume mechanism causes all further callbacks for the **HConn** to run on the thread where the MQCTL call is made. Therefore, the link between the **HConn** and the thread is not lost.

3.  On z/OS, when Operation is MQOP_START:

- Programs which use asynchronous callback routines must be authorized to use z/OS UNIX System Services (USS).
- Language Environment (LE) programs which use asynchronous callback routines must use the LE runtime option POSIX(ON).
- Non-LE programs which use asynchronous callback routines must not use the USS pthread_create interface (callable service BPX1PTC).

4.  MQCTL is not supported within the IMS adapter.

Note: In CICS, MQOP_START is not supported. Instead, use the MQOP_START_WAIT function call.

C invocation

```
MQCTL (Hconn, Operation, &ControlOpts, &CompCode, &Reason)
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQLONG   Operation;       /* Operation being processed */
MQCTLO   ControlOpts     /* Options that control the action of MQCTL */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQCTL' USING HCONN, OPERATION, CTLOPTS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN    PIC S9(9) BINARY.
** Operation
01 OPERATION PIC S9(9) BINARY.
** Control Options
01 CTLOPTS.
   COPY CMQCTLOV.
** Completion code
01 COMPCODE PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON   PIC S9(9) BINARY.
```

PL/I invocation

```
call MQCTL(Hconn, Operation, CtlOpts, CompCode, Reason)
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl Operation  fixed bin(31); /* Operation */
dcl CtlOpts like MQCTLO;      /* Options that control the action of MQCTL */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

MQDISC - Disconnect queue manager

The MQDISC call breaks the connection between the queue manager and the application program, and is the inverse of the MQCONN or MQCONNX call.


- On z/OS, all applications that use asynchronous message consumption, event handling or callback, the main control thread must issue an MQDISC call before ending. See [Asynchronous consumption of IBM MQ messages](#) for more details.
- On z/OS, CICS applications do not need to issue this call to disconnect from the queue manager.

If a CICS application does make this call it has no effect unless an earlier MQCONNX call was made, specifying one of the:

MQCNO_SERIALIZE_CONN_TAG_Q_MGR
MQCNO_SERIALIZE_CONN_TAG_QSG
MQCNO_RESTRICT_CONN_TAG_Q_MGR or
MQCNO_RESTRICT_CONN_TAG_QSG

options, in which case all currently open object handles are closed.

- On IBM i, applications running in compatibility mode do not need to issue this call. See [“MQCONN - Connect queue manager”](#) on page 634 for more information.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Syntax

MQDISC (*Hconn*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input/output

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, you can omit the MQCONN call, and specify the following value for *Hconn*:

MQHC_DEF_HCONN

Default connection handle.

On successful completion of the call, the queue manager sets *Hconn* to a value that is not a valid handle for the environment. This value is:

MQHC_UNUSABLE_HCONN

Unusable connection handle.

On z/OS, *Hconn* is set to a value that is undefined.

CompCode

Type: MQLONG - output

The completion code; it is one of the following codes:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_BACKED_OUT

(2003, X'7D3') Unit of work backed out.

MQRC_CONN_TAG_NOT_RELEASED

(2344, X'928') Connection tag not released.

MQRC_OUTCOME_PENDING

(2124, X'84C') Result of commit operation is pending.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_DISC_LOAD_ERROR

(2138, X'85A') Unable to load adapter disconnection module.

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_INIT_ERROR

(2375, X'947') API exit initialization failed.

MQRC_API_EXIT_TERM_ERROR

(2376, X'948') API exit termination failed.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_OUTCOME_MIXED

(2123, X'84B') Result of commit or back-out operation is mixed.

MQRC_PAGESET_ERROR

(2193, X'891') Error accessing page-set data set.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. If an MQDISC call is issued when the connection still has objects open under that connection, the queue manager closes those objects, with the close options set to MQCO_NONE.
2. If the application ends with uncommitted changes in a unit of work, the disposition of those changes depends on how the application ends:
 - a. If the application issues the MQDISC call before ending:
 - For a queue-manager-coordinated unit of work, the queue manager issues the MQCMIT call on behalf of the application. The unit of work is committed if possible, and backed out if not.
 - For an externally coordinated unit of work, there is no change in the status of the unit of work; however, the queue manager typically indicates that the unit of work must be committed when asked by the unit-of-work coordinator.

On z/OS, CICS, IMS (other than batch DL/1 programs), and RRS applications are like this.
 - b. If the application ends normally but without issuing the MQDISC call, the action taken depends on the environment:
 - On z/OS, except for MQ Java or MQ JMS applications, the actions described in note 2a occur.
 - In all other cases, the actions described in note 2c occur.

Because of the differences between environments, ensure that applications that you want to port either commit or back out the unit of work before they end.
 - c. If the application ends *abnormally* without issuing the MQDISC call, the unit of work is backed out.
3. On z/OS, the following points apply:
 - CICS applications do not have to issue the MQDISC call to disconnect from the queue manager, because the CICS system itself connects to the queue manager, and the MQDISC call has no effect on this connection.
 - CICS, IMS (other than batch DL/1 programs), and RRS applications use units of work that are coordinated by an external unit-of-work coordinator. As a result, the MQDISC call does not affect the status of the unit of work (if any) that exists when the call is issued.

However the MQDISC call *does* indicate the end of use of the connection tag *ConnTag* that was associated with the connection by an earlier MQCONN call issued by the application. If there is an active unit of work that references the connection tag when the MQDISC call is issued, the call completes with completion code MQCC_WARNING and reason code MQRC_CONN_TAG_NOT_RELEASED. The connection tag does not become available for reuse until the external unit-of-work coordinator has resolved the unit of work.
4. On IBM i, applications running in compatibility mode do not have to issue this call; see the MQCONN call for more details.

Note: In CICS, MQOP_START is not supported. Instead, use the MQOP_START_WAIT function call.

C invocation

```
MQDISC (&Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```

MQHCONN  Hconn;      /* Connection handle */
MQLONG   CompCode;   /* Completion code */
MQLONG   Reason;     /* Reason code qualifying CompCode */

```

COBOL invocation

```
CALL 'MQDISC' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```

** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.

```

PL/I invocation

```
call MQDISC (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```

dcl Hconn      fixed bin(31); /* Connection handle */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */

```

System/390 assembler invocation

```
CALL MQDISC,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```

HCONN      DS  F  Connection handle
COMPCODE   DS  F  Completion code
REASON     DS  F  Reason code qualifying COMPCODE

```

Visual Basic invocation

```
MQDISC Hconn, CompCode, Reason
```

Declare the parameters as follows:

```

Dim Hconn      As Long 'Connection handle'
Dim CompCode   As Long 'Completion code'
Dim Reason     As Long 'Reason code qualifying CompCode'

```

MQDLTMH - Delete message handle

The MQDLTMH call deletes a message handle and is the inverse of the MQCRTMH call.

Syntax

```
MQDLTMH (Hconn, Hmsg, DltMsgHOpts, CompCode, Reason)
```

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager.

The value must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using MQHC_UNASSOCIATED_HCONN then a valid connection must be established on the thread deleting the message handle, otherwise the call fails with MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMSG - input/output

This is the message handle to be deleted. The value was returned by a previous MQCRTMH call.

On successful completion of the call, the handle is set to an invalid value for the environment. This value is:

MQHM_UNUSABLE_HMSG

Unusable message handle.

The message handle cannot be deleted if another IBM MQ call is in progress that was passed the same message handle.

DltMsgHOpts

Type: MQDMHO - input

See [MQDMHO](#) for details.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CONNECTION_BROKEN

(2009, X'07D9') Connection to queue manager lost.

MQRC_DMHO_ERROR

(2462, X'099E') Delete message handle options structure not valid.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle pointer not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C invocation

```
MQDLTMH (Hconn, &Hmsg, &DltMsgHOpts, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;          /* Connection handle */
MQHMSG   Hmsg;           /* Message handle */
MQDMHO   DltMsgHOpts;    /* Options that control the action of MQDLTMH */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQDLTMH' USING HCONN, HMSG, DLTMSGOPTS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01  HCONN    PIC S9(9) BINARY.

** Options that control the action of MQDLTMH
01  DLTMSGOPTS.
COPY CMQDLMHOV.

** Completion code
01  COMPCODE  PIC S9(9) BINARY.

** Reason code qualifying COMPCODE
01  REASON    PIC S9(9) BINARY.
```

PL/I invocation

```
call MQDLTMH (Hconn, Hmsg, DltMsgHOpts, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn          /* Connection handle */
dcl Hmsg           /* Message handle */
dcl DltMsgHOpts like MQDMHO; /* Options that control the action of MQDLTMH */
dcl CompCode       /* Completion code */
dcl Reason         /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQDLTMH, (HCONN, HMSG, DLTMSGOPTS, COMPCODE, REASON)
```

Declare the parameters as follows:

```
HCONN      DS      F  Connection handle
HMSG       DS      D  Message handle
DLTMSGOPTS CMQDMHOA ,  Options that control the action of MQDLTMH
COMPCODE   DS      F  Completion code
REASON     DS      F  Reason code qualifying COMPCODE
```

MQDLTMP - Delete message property

The MQDLTMP call deletes a property from a message handle and is the inverse of the MQSETMP call.

Syntax

MQDLTMP (*Hconn*, *Hmsg*, *DltPropOpts*, *Name*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using MQHC_UNASSOCIATED_HCONN then a valid connection must be established on the thread deleting the message handle otherwise the call fails with MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMSG - input

This is the message handle containing the property to be deleted. The value was returned by a previous MQCRTMH call.

DltPropOpts

Type: MQDMPO - input

See the [MQDMPO](#) data type for details.

Name

Type: MQCHARV - input

The name of the property to delete. See [Property names](#) for further information about property names.

Wildcards are not allowed in the property name.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_PROPERTY_NOT_AVAILABLE

(2471, X'09A7') Property not available.

MQRC_RFH_FORMAT_ERROR

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'0852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'086D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CONNECTION_BROKEN

(2009, X'07D9') Connection to queue manager lost.

MQRC_DMPO_ERROR

(2481, X'09B1') Delete message property options structure not valid.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_PROPERTY_NAME_ERROR

(2442, X'098A') Invalid property name.

MQRC_SOURCE_CCSID_ERROR

(2111, X'083F') Property name coded character set identifier not valid.

MQRC_UNEXPECTED_ERROR

(2195, X'0893') Unexpected error occurred.

For detailed information about these codes, see:

- [Messages and reason codes for IBM MQ for z/OS](#)
- [API completion and reason codes for other IBM MQ platforms](#)

C invocation

```
MQDLTMP (Hconn, Hmsg, &DltPropOpts, &Name, &CompCode, &Reason)
```

Declare the parameters as follows:

```
MQHCONN Hconn;          /* Connection handle */
MQHMSG  Hmsg;            /* Message handle */
MQDMPO  DltPropOpts;    /* Options that control the action of MQDLTMP */
MQCHARV Name;           /* Property name */
MQLONG  CompCode;       /* Completion code */
MQLONG  Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQDLTMP' USING HCONN, HMSG, DLTPROPOPTS, NAME, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN    PIC S9(9) BINARY.
** Message handle
01 HMSG     PIC S9(18) BINARY.
** Options that control the action of MQDLTMP
```

```

01  DLTPROPOPTS.
    COPY CMQDMP0V.
**   Property name
01  NAME
    COPY CMQCHRVV.
**   Completion code
01  COMPCODE PIC S9(9) BINARY.
**   Reason code qualifying COMPCODE
01  REASON   PIC S9(9) BINARY.

```

PL/I invocation

```
call MQDLTMP (Hconn, Hmsg, DltPropOpts, Name, CompCode, Reason);
```

Declare the parameters as follows:

```

dcl Hconn      fixed bin(31); /* Connection handle */
dcl Hmsg       fixed bin(63); /* Message handle */
dcl DltPropOpts like MQDMP0; /* Options that control the action of MQDLTMP */
dcl Name       like MQCHARV; /* Property name */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```
CALL MQDLTMP, (HCONN,HMSG,DLTPROPOPTS,NAME,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HMSG	DS	D	Message handle
DLTPROPOPTS	CMQDMP0A	,	Options that control the action of MQDLTMP
NAME	CMQCHRVV	,	Property name
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQGET - Get message

The MQGET call retrieves a message from a local queue that has been opened using the MQOPEN call.

Syntax

```
MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer, DataLength, CompCode, Reason)
```

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Hobj

Type: MQHOBJ - input

This handle represents the queue from which a message is to be retrieved. The value of *Hobj* was returned by a previous MQOPEN call. The queue must have been opened with one or more of the following options (see [“MQOPEN - Open object”](#) on page 703 for details):

- MQOO_INPUT_SHARED
- MQOO_INPUT_EXCLUSIVE
- MQOO_INPUT_AS_Q_DEF
- MQOO_BROWSE

MsgDesc

Type: MQMD - input/output

This structure describes the attributes of the message required, and the attributes of the message retrieved. See [“MQMD - Message descriptor”](#) on page 396 for details.

If *BufferLength* is less than the message length, *MsgDesc* is filled by the queue manager, whether MQGMO_ACCEPT_TRUNCATED_MSG is specified on the *GetMsgOpts* parameter (see [MQGMO - Options field](#)).

If the application provides a version-1 MQMD, the message returned has an MQMDE prefixed to the application message data, but *only* if one or more of the fields in the MQMDE has a nondefault value. If all the fields in the MQMDE have default values, the MQMDE is omitted. A format name of MQFMT_MD_EXTENSION in the *Format* field in MQMD indicates that an MQMDE is present.

The application does not need to provide an MQMD structure if a valid message handle is supplied in the *MsgHandle* field. If nothing is provided in this field, the descriptor of the message is taken from the descriptor associated with the message handles.

If the application provides a message handle rather than an MQMD structure, and specifies MQGMO_PROPERTIES_FORCE_MQRFH2, the call fails with reason code MQRC_MD_ERROR. The call also fails, with reason code MQRC_MD_ERROR, if the application does not provide an MQMD structure and specifies MQGMO_PROPERTIES_AS_Q_DEF, and the *PropertyControl* queue attribute is MQPROP_FORCE_MQRFH2.

If match options are specified and the message descriptor associated with the message handle is being used, the input fields used for matching come from the message handle.

GetMsgOpts

Type: MQGMO - input/output

See [“MQGMO - Get-message options”](#) on page 350 for details.

BufferLength

Type: MQLONG - input

This is the length in bytes of the *Buffer* area. Specify zero for messages that have no data, or if the message is to be removed from the queue and the data discarded (you must specify MQGMO_ACCEPT_TRUNCATED_MSG in this case).

Note: The length of the longest message that it is possible to read from the queue is given by the *MaxMsgLength* queue attribute; see [“Attributes for queues”](#) on page 801.

Buffer

Type: MQBYTEExBufferLength - output

This is the area to contain the message data. Align the buffer on a boundary appropriate to the nature of the data in the message. 4 byte alignment is suitable for most messages (including messages containing IBM MQ header structures), but some messages might require more stringent alignment. For example, a message containing a 64 bit binary integer might require 8-byte alignment.

If *BufferLength* is less than the message length, as much of the message as possible is moved into *Buffer*; this happens whether MQGMO_ACCEPT_TRUNCATED_MSG is specified on the *GetMsgOpts* parameter (see [MQGMO - Options field](#) for more information).

The character set and encoding of the data in *Buffer* are given by the *CodedCharSetId* and *Encoding* fields returned in the *MsgDesc* parameter. If these values are different from the values required by the receiver, the receiver must convert the application message data to the character set and encoding required. The MQGMO_CONVERT option can be used (with a user-written exit if necessary) to convert the message data; see [“MQGMO - Get-message options”](#) on page 350 for details of this option.

Note: All the other parameters on the MQGET call are in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE).

If the call fails, the contents of the buffer might still have changed.

In the C programming language, the parameter is declared as a pointer-to-void: the address of any type of data can be specified as the parameter.

If the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

DataLength

Type: MQLONG - output

This is the length in bytes of the application data *in the message*. If this value is greater than *BufferLength*, only *BufferLength* bytes are returned in the *Buffer* parameter (that is, the message is truncated). If the value is zero, the message contains no application data.

If *BufferLength* is less than the message length, *DataLength* is still completed by the queue manager, whether MQGMO_ACCEPT_TRUNCATED_MSG is specified on the *GetMsgOpts* parameter (see [MQGMO - Options field](#) for more information). This allows the application to determine the size of the buffer required to accommodate the message data, and then reissue the call with a buffer of the appropriate size.

However, if the MQGMO_CONVERT option is specified, and the converted message data is too long to fit in *Buffer*, the value returned for *DataLength* is:

- The length of the *unconverted* data, for queue-manager defined formats.

In this case, if the nature of the data causes it to expand during conversion, the application must allocate a buffer bigger than the value returned by the queue manager for *DataLength*.

- The value returned by the data-conversion exit, for application-defined formats.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason codes listed are the ones that the queue manager can return for the *Reason* parameter. If the application specifies the MQGMO_CONVERT option, and a user-written exit is invoked to convert some or all the message data, the exit decides what value is returned for the *Reason* parameter. As a result, values other than those values documented are possible.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQR_CONVERTED_MSG_TOO_BIG
(2120, X'848') Converted data too large for buffer.

MQR_CONVERTED_STRING_TOO_BIG
(2190, X'88E') Converted string too large for field.

MQR_DBCS_ERROR
(2150, X'866') DBCS string not valid.

MQR_FORMAT_ERROR
(2110, X'83E') Message format not valid.

MQR_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.

MQR_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.

MQR_INCONSISTENT_CCIDS
(2243, X'8C3') Message segments have differing CCIDs.

MQR_INCONSISTENT_ENCODINGS
(2244, X'8C4') Message segments have differing encodings.

MQR_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.

MQR_MSG_TOKEN_ERROR
(2331, X'91B') Invalid use of message token.

MQR_NO_MSG_LOCKED
(2209, X'8A1') No message locked.

MQR_NOT_CONVERTED
(2119, X'847') Message data not converted.

MQR_OPTIONS_CHANGED
(nnnn, X'xxx') Options that were required to be consistent have been changed.

MQR_PARTIALLY_CONVERTED
(2272, X'8E0') Message data partially converted.

MQR_SIGNAL_REQUEST_ACCEPTED
(2070, X'816') No message returned (but signal request accepted).

MQR_SOURCE_BUFFER_ERROR
(2145, X'861') Source buffer parameter not valid.

MQR_SOURCE_CCID_ERROR
(2111, X'83F') Source coded character set identifier not valid.

MQR_SOURCE_DECIMAL_ENC_ERROR
(2113, X'841') Packed-decimal encoding in message not recognized.

MQR_SOURCE_FLOAT_ENC_ERROR
(2114, X'842') Floating-point encoding in message not recognized.

MQR_SOURCE_INTEGER_ENC_ERROR
(2112, X'840') Source integer encoding not recognized.

MQR_SOURCE_LENGTH_ERROR
(2143, X'85F') Source length parameter not valid.

MQR_TARGET_BUFFER_ERROR
(2146, X'862') Target buffer parameter not valid.

MQR_TARGET_CCID_ERROR
(2115, X'843') Target coded character set identifier not valid.

MQR_TARGET_DECIMAL_ENC_ERROR
(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

MQR_TARGET_FLOAT_ENC_ERROR
(2118, X'846') Floating-point encoding specified by receiver not recognized.

MQRC_TARGET_INTEGER_ENC_ERROR

(2116, X'844') Target integer encoding not recognized.

MQRC_TRUNCATED_MSG_ACCEPTED

(2079, X'81F') Truncated message returned (processing completed).

MQRC_TRUNCATED_MSG_FAILED

(2080, X'820') Truncated message returned (processing not completed).

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_CONV_LOAD_ERROR

(2133, X'855') Unable to load data conversion services modules.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BACKED_OUT

(2003, X'7D3') Unit of work backed out.

MQRC_BUFFER_ERROR

(2004, X'7D4') Buffer parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') Coupling facility not available.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CF_STRUC_LIST_HDR_IN_USE

(2347, X'92B') Coupling-facility structure list-header in use.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CORREL_ID_ERROR

(2207, X'89F') Correlation-identifier error.

MQRC_DATA_LENGTH_ERROR

(2010, X'7DA') Data length parameter not valid.

MQRC_DB2_NOT_AVAILABLE
(2342, X'926') Db2 subsystem not available.

MQRC_GET_INHIBITED
(2016, X'7E0') Gets inhibited for the queue.

MQRC_GLOBAL_UOW_CONFLICT
(2351, X'92F') Global units of work conflict.

MQRC_GMO_ERROR
(2186, X'88A') Get-message options structure not valid.

MQRC_HANDLE_IN_USE_FOR_UOW
(2353, X'931') Handle in use for global unit of work.

MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.

MQRC_INCONSISTENT_BROWSE
(2259, X'8D3') Inconsistent browse specification.

MQRC_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.

MQRC_INVALID_MSG_UNDER_CURSOR
(2246, X'8C6') Message under cursor not valid for retrieval.

MQRC_LOCAL_UOW_CONFLICT
(2352, X'930') Global unit of work conflicts with local unit of work.

MQRC_MATCH_OPTIONS_ERROR
(2247, X'8C7') Match options not valid.

MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.

MQRC_MSG_ID_ERROR
(2206, X'89E') Message-identifier error.

MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.

MQRC_MSG_TOKEN_ERROR
(2331, X'91B') Use of message token not valid.

MQRC_NO_MSG_AVAILABLE
(2033, X'7F1') No message available.

MQRC_NO_MSG_UNDER_CURSOR
(2034, X'7F2') Browse cursor not positioned on message.

MQRC_NOT_OPEN_FOR_BROWSE
(2036, X'7F4') Queue not open for browse.

MQRC_NOT_OPEN_FOR_INPUT
(2037, X'7F5') Queue not open for input.

MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.

MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.

MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page-set data set.

MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.

MQRC_Q_INDEX_TYPE_ERROR

(2394, X'95A') Queue has wrong index type.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SECOND_MARK_NOT_ALLOWED

(2062, X'80E') A message is already marked.

MQRC_SIGNAL_OUTSTANDING

(2069, X'815') Signal outstanding for this handle.

MQRC_SIGNAL1_ERROR

(2099, X'833') Signal field not valid.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_SYNCPOINT_LIMIT_REACHED

(2024, X'7E8') No more messages can be handled within current unit of work.

MQRC_SYNCPOINT_NOT_AVAILABLE

(2072, X'818') sync point support not available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UOW_ENLISTMENT_ERROR

(2354, X'932') Enlistment in global unit of work failed.

MQRC_UOW_MIX_NOT_SUPPORTED

(2355, X'933') Mixture of unit-of-work calls not supported.

MQRC_UOW_NOT_AVAILABLE

(2255, X'8CF') Unit of work not available for the queue manager to use.

MQRC_WAIT_INTERVAL_ERROR

(2090, X'82A') Wait interval in MQGMO not valid.

MQRC_WRONG_GMO_VERSION

(2256, X'8D0') Wrong version of MQGMO supplied.

MQRC_WRONG_MD_VERSION

(2257, X'8D1') Wrong version of MQMD supplied.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. The message retrieved is normally deleted from the queue. This deletion can occur as part of the MQGET call itself, or as part of a sync point.

The browse options are: MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT, and MQGMO_BROWSE_MSG_UNDER_CURSOR.

2. If the MQGMO_LOCK option is specified with one of the browse options, the browsed message is locked so that it is visible only to this handle.

If the MQGMO_UNLOCK option is specified, a previously locked message is unlocked. No message is retrieved in this case, and the *MsgDesc*, *BufferLength*, *Buffer*, and *DataLength* parameters are not checked or altered.

3. For applications issuing an MQGET call, the message retrieved can be lost if the application terminates abnormally or the connection is severed while processing the call. This issue arises because the surrogate running on the same platform as the queue manager that issues the MQGET call on behalf of the application cannot detect the loss of the application until the surrogate is about to return the message to the application, *after* the message has been removed from the queue. This issue can occur for both persistent messages and nonpersistent messages.

To eliminate the risk of losing messages in this way, always retrieve messages within units of work. That is, by specifying the MQGMO_SYNCPOINT option on the MQGET call, and using the MQCMIT or MQBACK calls to commit or back out the unit of work when message processing is complete. If MQGMO_SYNCPOINT is specified, and the client terminates abnormally or the connection is severed, the surrogate backs out the unit of work on the queue manager and the message is reinstated on the queue. For more information about sync points, see [Syncpoint considerations in IBM MQ applications](#).

This situation can arise with IBM MQ clients as well as with applications that are running on the same platform as the queue-manager.

4. If an application puts a sequence of messages on a particular queue within a single unit of work, and then commits that unit of work successfully, the messages become available for retrieval as follows:
 - If the queue is a *nonshared* queue (that is, a local queue), all messages within the unit of work become available at the same time.
 - If the queue is a *shared* queue, messages within the unit of work become available in the order in which they were put, but not all at the same time. When the system is heavily laden, it is possible for the first message in the unit of work to be retrieved successfully, but for the MQGET call for the second or subsequent message in the unit of work to fail with MQRC_NO_MSG_AVAILABLE. If this issue occurs, the application must wait a short while and then try the operation again.
5. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if certain conditions are satisfied. See [MQPUT usage notes](#) for details. If the conditions are satisfied, the messages are presented to the receiving application in the order in which they were sent, if:
 - Only one receiver is getting messages from the queue.

If there are two or more applications getting messages from the queue, they must agree with the sender the mechanism to be used to identify messages that belong to a sequence. For example, the sender might set all the *CorrelId* fields in the messages in a sequence to a value that was unique to that sequence of messages.

- The receiver does not deliberately change the order of retrieval, for example by specifying a particular *MsgId* or *CorrelId*.

If the sending application puts the messages as a message group, the messages are presented to the receiving application in the correct order if the receiving application specifies the MQGMO_LOGICAL_ORDER option on the MQGET call. For more information about message groups, see:

- [MQMD - MsgFlags field](#)
- [MQPMO_LOGICAL_ORDER](#)
- [MQGMO_LOGICAL_ORDER](#)

If the user is getting messages in a group under sync point, they must ensure that the complete group is processed before attempting to finish the transaction.

6. Applications must test for the feedback code MQFB_QUIT in the *Feedback* field of the *MsgDesc* parameter, and end if they find this value. See [MQMD - Feedback field](#) for more information.
7. If the queue identified by *Hobj* was opened with the MQOO_SAVE_ALL_CONTEXT option, and the completion code from the MQGET call is MQCC_OK or MQCC_WARNING, the context associated with the queue handle *Hobj* is set to the context of the message that has been retrieved (unless the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT, or MQGMO_BROWSE_MSG_UNDER_CURSOR option is set, in which case the context is marked as not available).

You can use the saved context on a subsequent MQPUT or MQPUT1 call by specifying the MQPMO_PASS_IDENTITY_CONTEXT or MQPMO_PASS_ALL_CONTEXT options. This enables the context of the message received to be transferred in whole or in part to another message (for example, when the message is forwarded to another queue). For more information about message context, see [Message context](#).

8. If you include the MQGMO_CONVERT option in the *GetMsgOpts* parameter, the application message data is converted to the representation requested by the receiving application, before the data is placed in the *Buffer* parameter:
 - The *Format* field in the control information in the message identifies the structure of the application data, and the *CodedCharSetId* and *Encoding* fields in the control information in the message specify its character-set identifier and encoding.
 - The application issuing the MQGET call specifies in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter the character-set identifier and encoding to which to convert the application message data.

When conversion of the message data is necessary, the conversion is performed either by the queue manager itself or by a user-written exit, depending on the value of the *Format* field in the control information in the message:

- The following format names are formats that are converted by the queue manager; these formats are called "built-in" formats:
 - MQFMT_ADMIN
 - MQFMT_CICS (z/OS only)
 - MQFMT_COMMAND_1
 - MQFMT_COMMAND_2
 - MQFMT_DEAD_LETTER_HEADER
 - MQFMT_DIST_HEADER
 - MQFMT_EVENT version 1
 - MQFMT_EVENT version 2 (z/OS only)
 - MQFMT_IMS
 - MQFMT_IMS_VAR_STRING
 - MQFMT_MD_EXTENSION
 - MQFMT_PCF
 - MQFMT_REF_MSG_HEADER
 - MQFMT_RF_HEADER
 - MQFMT_RF_HEADER_2
 - MQFMT_STRING
 - MQFMT_TRIGGER
 - MQFMT_WORK_INFO_HEADER (z/OS only)
 - MQFMT_XMIT_Q_HEADER
- The format name MQFMT_NONE is a special value that indicates that the nature of the data in the message is undefined. As a consequence, the queue manager does not attempt conversion when the message is retrieved from the queue.

Note: If MQGMO_CONVERT is specified on the MQGET call for a message that has a format name of MQFMT_NONE, and the character set or encoding of the message differs from that specified in the *MsgDesc* parameter, the message is returned in the *Buffer* parameter (assuming no other errors), but the call completes with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

You can use MQFMT_NONE either when the nature of the message data means that it does not require conversion, or when the sending and receiving applications have agreed between themselves the form in which to send the message data.

- All other format names pass the message to a user-written exit for conversion. The exit has the same name as the format, apart from environment-specific additions. User-specified format names must not begin with the letters IBM MQ.

See [“Data-conversion exit” on page 870](#) for details of the data-conversion exit.

User data in the message can be converted between any supported character sets and encodings. However, be aware that, if the message contains one or more IBM MQ header structures, the message cannot be converted from or to a character set that has double-byte or multi-byte characters for any of the characters that are valid in queue names. Reason code MQRC_SOURCE_CCSD_ERROR or MQRC_TARGET_CCSD_ERROR results if this is attempted, and the message is returned unconverted. Unicode character set UCS-2 is an example of such a character set.

On return from MQGET, the following reason code indicates that the message was converted successfully:

- MQRC_NONE

The following reason code indicates that the message *might* have been converted successfully; the application must check the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to find out:

- MQRC_TRUNCATED_MSG_ACCEPTED

All other reason codes indicate that the message was not converted.

Note: The interpretation of this reason code is true for conversions performed by a user-written exit *only* if the exit conforms to the processing guidelines described in [“Data-conversion exit” on page 870](#).

9. When using the object-oriented interface to get messages, you can choose not to specify a buffer to hold the message data for an MQGET call. However, in versions of IBM MQ, prior to version 7, it was possible for MQGET to fail with reason code MQRC_CONVERTED_MSG_TO_BIG, even when a buffer was not specified. From IBM MQ Version 7, when you get a message using an object-oriented application without restricting the size of the receive message buffer, the application does not fail with MQRC_CONVERTED_MSG_TOO_BIG, and receives the converted message. This is true of the following environments:

- .NET, including fully managed applications
- C++
- Java (IBM MQ classes for Java)

Note: For all clients, if the value of *sharingConversations* is zero, the channel operates as it did before IBM WebSphere MQ 7.0, and message handling reverts to Version 6 behavior. In this situation, if the buffer is too small to receive the converted message, the unconverted message is returned, with reason code MQRC_CONVERTED_MSG_TOO_BIG. For more information about *sharingConversations*, see [Using sharing conversations in a client application](#).

10. For the built-in formats, the queue manager can perform *default conversion* of character strings in the message when the MQGMO_CONVERT option is specified. Default conversion allows the queue manager to use an installation-specified default character set that approximates the actual character set, when converting string data. As a result, the MQGET call can succeed

with completion code MQCC_OK, instead of completing with MQCC_WARNING and reason code MQRC_SOURCE_CCSD_ERROR or MQRC_TARGET_CCSD_ERROR.

Note: The result of using an approximate character set to convert string data is that some characters might be converted incorrectly. To avoid this, use characters in the string that are common to both the actual character set and the default character set.

Default conversion applies both to the application message data and to character fields in the MQMD and MQMDE structures:

- Default conversion of the application message data occurs only when *all* the following are true:
 - The application specifies MQGMO_CONVERT.
 - The message contains data that must be converted either from or to a character set that is not supported.
 - Default conversion was enabled when the queue manager was installed or restarted.
- Default conversion of the character fields in the MQMD and MQMDE structures occurs as necessary, if default conversion is enabled for the queue manager. The conversion is performed even if the MQGMO_CONVERT option is not specified by the application on the MQGET call.

11. For the Visual Basic programming language, the following points apply:

- If the size of the *Buffer* parameter is less than the length specified by the *BufferLength* parameter, the call fails with reason code MQRC_STORAGE_NOT_AVAILABLE.
- The *Buffer* parameter is declared as being of type String. If the data to be retrieved from the queue is not of type String, use the MQGETAny call in place of MQGET.

The MQGETAny call has the same parameters as the MQGET call, except that the *Buffer* parameter is declared as being of type Any, allowing any type of data to be retrieved. However, this means that *Buffer* cannot be checked to ensure that it is at least *BufferLength* bytes in size.

12. Not all MQGET options are supported when read ahead is enabled. The following table indicated which options are allowed and whether they can be altered between MQGET calls.

Table 554. MQGET options permitted when read ahead is enabled			
	Permitted when read ahead is enabled and can be altered between MQGET calls	Permitted when read ahead is enabled but cannot be altered between MQGET calls ^a	MQGET options that are not permitted when read ahead is enabled ^b
MQGET MD values	MsgId ^c CorrelId ^c	Encoding CodedCharSetId	
MQGET MQGMO options	MQGMO_WAIT MQGMO_NO_WAIT MQGMO_FAIL_IF_QUIESCING MQGMO_BROWSE_FIRST ^d MQGMO_BROWSE_NEXT ^d MQGMO_BROWSE_MESSAGE _UNDER_CURSOR ^d	MQGMO_SYNCPOINT_IF_PERSISTENT MQGMO_NO_SYNCPOINT MQGMO_ACCEPT_TRUNCATED_MSG MQGMO_CONVERT MQGMO_LOGICAL_ORDER MQGMO_COMPLETE_MSG MQGMO_ALL_MSGS_AVAILABLE MQGMO_ALL_SEGMENTS_AVAILABLE MQGMO_MARK_BROWSE_HANDLE MQGMO_UNMARK_BROWSE_CO_OP MQGMO_UNMARK_BROWSE_HANDLE MQGMO_UNMARKED_BROWSE_MSG MQGMO_PROPERTIES_FORCE_MQRFH2 MQGMO_NO_PROPERTIES MQGMO_PROPERTIES_IN_HANDLE MQGMO_PROPERTIES_COMPATIBILITY	MQGMO_SET_SIGNAL MQGMO_SYNCPOINT MQGMO_MARK_SKIP _BACKOUT MQGMO_MSG_UNDER _CURSOR ^d MQGMO_LOCK MQGMO_UNLOCK
MQGMO values		MsgHandle	

- If these options are altered between MQGET calls an MQRC_OPTIONS_CHANGED reason code is returned.
- If these options are specified on the first MQGET call then read ahead is disabled. If these options are specified on a subsequent MQGET call a reason code MQRC_OPTIONS_ERROR is returned.
- The client applications need to be aware that if the MsgId and CorrelId values are altered between MQGET calls messages with the previous values might have already been sent to the client and remain in the client read ahead buffer until consumed (or automatically purged).

- d. The first MQGET call determines whether messages are to be browsed or got from a queue when read ahead is enabled. If the application attempts to use a combination of browse and get an MQRC_OPTIONS_CHANGED reason code is returned.
 - e. MQGMO_MSG_UNDER_CURSOR is not possible with read ahead. Messages can be browsed or got when read ahead is enabled but not a combination of both.
13. Applications can destructively get uncommitted messages only if those messages are put in the same local unit of work as the get. Applications cannot get uncommitted messages nondestructively.
 14. Messages under a browse cursor can be retrieved in a unit of work. It is not possible to retrieve an uncommitted message in this way.

C invocation

```
MQGET (Hconn, Hobj, &MsgDesc, &GetMsgOpts, BufferLength, Buffer,
      &DataLength, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQHOBJ   Hobj;            /* Object handle */
MQMD     MsgDesc;         /* Message descriptor */
MQGMO    GetMsgOpts;      /* Options that control the action of MQGET */
MQLONG   BufferLength;     /* Length in bytes of the Buffer area */
MQBYTE   Buffer[n];       /* Area to contain the message data */
MQLONG   DataLength;      /* Length of the message */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQGET' USING HCONN, HOBJ, MSGDESC, GETMSGOPTS, BUFFERLENGTH,
BUFFER, DATALENGTH, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Object handle
01 HOBJ           PIC S9(9) BINARY.
** Message descriptor
01 MSGDESC.
   COPY CMQMDV.
** Options that control the action of MQGET
01 GETMSGOPTS.
   COPY CMQGMOV.
** Length in bytes of the BUFFER area
01 BUFFERLENGTH  PIC S9(9) BINARY.
** Area to contain the message data
01 BUFFER        PIC X(n).
** Length of the message
01 DATALENGTH   PIC S9(9) BINARY.
** Completion code
01 COMPCODE      PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON        PIC S9(9) BINARY.
```

PL/I invocation

```
call MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer,
            DataLength, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl Hobj       fixed bin(31); /* Object handle */
dcl MsgDesc    like MQMD;     /* Message descriptor */
dcl GetMsgOpts like MQGMO;     /* Options that control the action of
```

```

MQGET */
dcl BufferLength    fixed bin(31); /* Length in bytes of the Buffer
                                area */
dcl Buffer          char(n);      /* Area to contain the message data */
dcl DataLength     fixed bin(31); /* Length of the message */
dcl CompCode       fixed bin(31); /* Completion code */
dcl Reason         fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```

CALL MQGET, (HCONN,HOBJ,MSGDESC,GETMSGOPTS,BUFFERLENGTH,
            BUFFER,DATALength,COMPCode,Reason)

```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HOBJ	DS	F	Object handle
MSGDESC	CMQMDA	,	Message descriptor
GETMSGOPTS	CMQGMOA	,	Options that control the action of MQGET
BUFFERLENGTH	DS	F	Length in bytes of the BUFFER area
BUFFER	DS	CL(n)	Area to contain the message data
DATALength	DS	F	Length of the message
COMPCode	DS	F	Completion code
Reason	DS	F	Reason code qualifying COMPCode

Visual Basic invocation

```

MQGET Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer,
DataLength, CompCode, Reason

```

Declare the parameters as follows:

Dim Hconn	As Long	'Connection handle'
Dim Hobj	As Long	'Object handle'
Dim MsgDesc	As MQMD	'Message descriptor'
Dim GetMsgOpts	As MQGMO	'Options that control the action of MQGET'
Dim BufferLength	As Long	'Length in bytes of the Buffer area'
Dim Buffer	As String	'Area to contain the message data'
Dim DataLength	As Long	'Length of the message'
Dim CompCode	As Long	'Completion code'
Dim Reason	As Long	'Reason code qualifying CompCode'

MQINQ - Inquire object attributes

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object.

The following types of object are valid:

- Queue manager
- Queue
- Namelist
- Process definition

Syntax

```

MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs, CharAttrLength, CharAttrs,
CompCode, Reason)

```

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Hobj

Type: MQHOBJ - input

This handle represents the object (of any type) with attributes that are required. The handle must be returned by a previous MQOPEN call that specified the MQOO_INQUIRE option.

SelectorCount

Type: MQLONG - input

This is the count of selectors that are supplied in the *Selectors* array. It is the number of attributes that are to be returned. Zero is a valid value. The maximum number allowed is 256.

Selectors

Type: MQLONG x *SelectorCount* - input

This is an array of *SelectorCount* attribute selectors; each selector identifies an attribute (integer or character) with a value that is required.

Each selector must be valid for the type of object that *Hobj* represents, otherwise the call fails with completion code MQCC_FAILED and reason code MQRC_SELECTOR_ERROR.

In the special case of queues:

- If the selector is not valid for queues of any type, the call fails with completion code MQCC_FAILED and reason code MQRC_SELECTOR_ERROR.
- If the selector applies only to queues of types other than the type of the object, the call succeeds with completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE.
- If the queue being inquired is a cluster queue, the selectors that are valid depend on how the queue was resolved; see “Usage notes” on page 691 for further details.

You can specify selectors in any order. Attribute values that correspond to integer attribute selectors (MQIA_* selectors) are returned in *IntAttrs* in the same order in which these selectors occur in *Selectors*. Attribute values that correspond to character attribute selectors (MQCA_* selectors) are returned in *CharAttrs* in the same order in which those selectors occur. MQIA_* selectors can be interleaved with the MQCA_* selectors; only the relative order within each type is important.

Note:

1. The integer and character attribute selectors are allocated within two different ranges; the MQIA_* selectors reside within the range MQIA_FIRST through MQIA_LAST, and the MQCA_* selectors within the range MQCA_FIRST through MQCA_LAST.

For each range, the constants MQIA_LAST_USED and MQCA_LAST_USED define the highest value that the queue manager accepts.

2. If all the MQIA_* selectors occur first, the same element numbers can be used to address corresponding elements in the *Selectors* and *IntAttrs* arrays.
3. If the *SelectorCount* parameter is zero, *Selectors* is not referred to. In this case, the parameter address passed by programs written in C or S/390 assembler might be null.

The attributes that can be inquired are listed in the following tables. For the MQCA_* selectors, the constant that defines the length in bytes of the resulting string in *CharAttrs* is provided in parentheses.

The tables that follow list the selectors, by object, in alphabetical order, as follows:

- [Table 555 on page 679](#) MQINQ attribute selectors for queues
- [Table 556 on page 681](#) MQINQ attribute selectors for namelists
- [Table 557 on page 682](#) MQINQ attribute selectors for process definitions
- [Table 558 on page 682](#) MQINQ attribute selectors for the queue manager

All selectors are supported on all IBM MQ platforms, except where indicated in the **Note** column as follows:

Not z/OS

Supported on all platforms **except** z/OS

z/OS

Supported **only** on z/OS

Table 555. MQINQ attribute selectors for queues			
Selector	Length of field	Description	Note
MQCA_ALTERATION_DATE	MQ_DATE_LENGTH	Date of most-recent alteration	
MQCA_ALTERATION_TIME	MQ_TIME_LENGTH	Time of most-recent alteration	
MQCA_BACKOUT_REQ_Q_NAME	MQ_Q_NAME_LENGTH	Excessive backout requeue name	
MQCA_BASE_Q_NAME	MQ_Q_NAME_LENGTH	Name of queue that alias resolves to	
MQCA_CF_STRUC_NAME	MQ_CF_STRUC_NAME_LENGTH	Coupling-facility structure name	z/OS
MQCA_CLUS_CHL_NAME	MQ_CHANNEL_NAME_LENGTH	Name of the cluster-sender channel that uses this queue as a transmission queue.	
MQCA_CLUSTER_NAME	MQ_CLUSTER_NAME_LENGTH	Cluster name	
MQCA_CLUSTER_NAMELIST	MQ_NAMELIST_NAME_LENGTH	Cluster namelist	
MQCA_CREATION_DATE	MQ_CREATION_DATE_LENGTH	Queue creation date	
MQCA_CREATION_TIME	MQ_CREATION_TIME_LENGTH	Queue creation time	
MQCA_CUSTOM	MQ_CUSTOM_LENGTH	The custom attribute for new features	
MQCA_INITIATION_Q_NAME	MQ_Q_NAME_LENGTH	Initiation queue name	
MQCA_PROCESS_NAME	MQ_PROCESS_NAME_LENGTH	Name of process definition	
MQCA_Q_DESC	MQ_Q_DESC_LENGTH	Queue description	
MQCA_Q_NAME	MQ_Q_NAME_LENGTH	Queue name	
MQCA_REMOTE_Q_MGR_NAME	MQ_Q_MGR_NAME_LENGTH	Name of remote queue manager	
MQCA_REMOTE_Q_NAME	MQ_Q_NAME_LENGTH	Name of remote queue as known on remote queue manager	

<i>Table 555. MQINQ attribute selectors for queues (continued)</i>			
Selector	Length of field	Description	Note
MQCA_STORAGE_CLASS	MQ_STORAGE_CLASS_LENGTH	Name of storage class	z/OS
MQCA_TRIGGER_DATA	MQ_TRIGGER_DATA_LENGTH	Trigger data	
MQCA_XMIT_Q_NAME	MQ_Q_NAME_LENGTH	Transmission queue name	
MQIA_ACCOUNTING_Q	MQLONG	Controls collection of accounting data for queue	Not z/OS
MQIA_BACKOUT_THRESHOLD	MQLONG	Backout threshold	
MQIA_CLWL_Q_PRIORITY	MQLONG	Priority of queue	
MQIA_CLWL_Q_RANK	MQLONG	Rank of queue	
MQIA_CLWL_USEQ	MQLONG	Use remote queues	
MQIA_CURRENT_Q_DEPTH	MQLONG	Number of messages on queue	
MQIA_DEF_BIND	MQLONG	Default binding	
MQIA_DEF_INPUT_OPEN_OPTION	MQLONG	Default open-for-input option	
MQIA_DEF_PERSISTENCE	MQLONG	Default message persistence	
MQIA_DEF_PRIORITY	MQLONG	Default message priority	
MQIA_DEFINITION_TYPE	MQLONG	Queue definition type	
MQIA_DIST_LISTS	MQLONG	Distribution list support	Not z/OS
MQIA_HARDEN_GET_BACKOUT	MQLONG	Whether to harden backout count	
MQIA_INDEX_TYPE	MQLONG	Type of index maintained for queue	z/OS
MQIA_INHIBIT_GET	MQLONG	Whether get operations are allowed	
MQIA_INHIBIT_PUT	MQLONG	Whether put operations are allowed	
MQIA_MAX_MSG_LENGTH	MQLONG	Maximum message length	
MQIA_MAX_Q_DEPTH	MQLONG	Maximum number of messages allowed on queue	
MQIA_MSG_DELIVERY_SEQUENCE	MQLONG	Whether message priority is relevant	
MQIA_NPM_CLASS	MQLONG	Level of reliability for nonpersistent messages	
MQIA_OPEN_INPUT_COUNT	MQLONG	Number of MQOPEN calls that have the queue open for input	
MQIA_OPEN_OUTPUT_COUNT	MQLONG	Number of MQOPEN calls that have the queue open for output	
MQIA_PROPERTY_CONTROL	MQLONG	Property control attribute	
MQIA_Q_DEPTH_HIGH_EVENT	MQLONG	Control attribute for queue depth high events	Not z/OS

<i>Table 555. MQINQ attribute selectors for queues (continued)</i>			
Selector	Length of field	Description	Note
MQIA_Q_DEPTH_HIGH_LIMIT	MQLONG	High limit for queue depth	Not z/OS
MQIA_Q_DEPTH_LOW_EVENT	MQLONG	Control attribute for queue depth low events	Not z/OS
MQIA_Q_DEPTH_LOW_LIMIT	MQLONG	Low limit for queue depth	Not z/OS
MQIA_Q_DEPTH_MAX_EVENT	MQLONG	Control attribute for queue depth max events	Not z/OS
MQIA_Q_SERVICE_INTERVAL	MQLONG	Limit for queue service interval	Not z/OS
MQIA_Q_SERVICE_INTERVAL_EVENT	MQLONG	Control attribute for queue service interval events	Not z/OS
MQIA_Q_TYPE	MQLONG	Queue type	
MQIA_QSG_DISP	MQLONG	Queue-sharing group disposition	z/OS
MQIA_RETENTION_INTERVAL	MQLONG	Queue retention interval	
MQIA_SCOPE	MQLONG	Queue definition scope	Not z/OS
MQIA_SHAREABILITY	MQLONG	Whether queue can be shared for input	
MQIA_STATISTICS_Q	MQLONG	Controls collection of statistics data for queue	Not z/OS
MQIA_TRIGGER_CONTROL	MQLONG	Trigger control	
MQIA_TRIGGER_DEPTH	MQLONG	Trigger depth	
MQIA_TRIGGER_MSG_PRIORITY	MQLONG	Threshold message priority for triggers	
MQIA_TRIGGER_TYPE	MQLONG	Trigger type	
MQIA_USAGE	MQLONG	Usage	

<i>Table 556. MQINQ attribute selectors for namelists</i>			
Selector	Length of field	Description	Note
MQCA_ALTERATION_DATE	MQ_DATE_LENGTH	Date of most-recent alteration	
MQCA_ALTERATION_TIME	MQ_TIME_LENGTH	Time of most-recent alteration	
MQCA_NAMELIST_DESC	MQ_NAMELIST_DESC_LENGTH	Namelist description	
MQCA_NAMELIST_NAME	MQ_NAMELIST_NAME_LENGTH	Name of namelist object	
MQIA_NAMELIST_TYPE	MQLONG	Namelist type	z/OS
MQCA_NAMES	MQ_Q_NAME_LENGTH x <i>Number of names in the list</i>	Names in the namelist	

Table 556. MQINQ attribute selectors for namelists (continued)

Selector	Length of field	Description	Note
MQIA_NAME_COUNT	MLONG	Number of names in the namelist	
MQIA_QSG_DISP	MLONG	Queue-sharing group disposition	z/OS

Table 557. MQINQ attribute selectors for process definitions

Selector	Length of field	Description	Note
MQCA_ALTERATION_DATE	MQ_DATE_LENGTH	Date of most-recent alteration	
MQCA_ALTERATION_TIME	MQ_TIME_LENGTH	Time of most-recent alteration	
MQCA_APPL_ID	MQ_PROCESS_APPL_ID_LENGTH	Application identifier	
MQCA_ENV_DATA	MQ_PROCESS_ENV_DATA_LENGTH	Environment data	
MQCA_PROCESS_DESC	MQ_PROCESS_DESC_LENGTH	Description of process definition	
MQCA_PROCESS_NAME	MQ_PROCESS_NAME_LENGTH	Name of process definition	
MQCA_USER_DATA	MQ_PROCESS_USER_DATA_LENGTH	User data	
MQIA_APPL_TYPE	MLONG	Application type	
MQIA_QSG_DISP	MLONG	Queue-sharing group disposition	z/OS

Table 558. MQINQ attribute selectors for the queue manager

Selector	Length of field	Description	Note
MQCA_ALTERATION_DATE	MQ_DATE_LENGTH	Date of most-recent alteration	
MQCA_ALTERATION_TIME	MQ_TIME_LENGTH	Time of most-recent alteration	
MQCA_CHANNEL_AUTO_DEF_EXIT	MQ_EXIT_NAME_LENGTH	Automatic channel definition exit name	
MQCA_CHINIT_SERVICE_PARM		Reserved for use by IBM	
MQCA_CLUSTER_WORKLOAD_DATA	MQ_EXIT_DATA_LENGTH	Data passed to cluster workload exit	
MQCA_CLUSTER_WORKLOAD_EXIT	MQ_EXIT_NAME_LENGTH	Name of cluster workload exit	
MQCA_COMMAND_INPUT_QUEUE_NAME	MQ_Q_NAME_LENGTH	System command input queue name	
MQCA_CUSTOM	MQ_CUSTOM_LENGTH	The custom attribute for new features	
MQCA_DEAD_LETTER_QUEUE_NAME	MQ_Q_NAME_LENGTH	Name of dead-letter queue	
MQCA_DEF_XMIT_QUEUE_NAME	MQ_Q_NAME_LENGTH	Default transmission queue name	

Table 558. MQINQ attribute selectors for the queue manager (continued)

Selector	Length of field	Description	Note
MQCA_DNS_GROUP	MQ_DNS_GROUP_NAME_LENGTH	Name of the group for the TCP listener that handles inbound transmissions for the queue-sharing group to join. The name applies when using Workload Manager Dynamic Domain Name Services.	z/OS
MQCA_IGQ_USER_ID	MQ_USER_ID_LENGTH	Intra-group queuing user identifier	z/OS
MQCA_INSTALLATION_DESC	MQ_INSTALLATION_DESC_LENGTH	Description of the associated installation	Not z/OS . Not IBM i
MQCA_INSTALLATION_NAME	MQ_INSTALLATION_NAME_LENGTH	Name of the installation associated with the queue manager	Not z/OS . Not IBM i
MQCA_INSTALLATION_PATH	MQ_INSTALLATION_PATH_LENGTH	Path where the associated IBM MQ is installed	Not z/OS . Not IBM i
MQCA_LU_GROUP_NAME	MQ_LU_NAME_LENGTH	Generic LU name for the LU 6.2 listener that handles inbound transmissions for the queue-sharing group to use	z/OS
MQCA_LU_NAME	MQ_LU_NAME_LENGTH	Name of the LU to use for outbound LU 6.2 transmissions. Set this name to the same LU that the listener uses for inbound transmissions	z/OS
MQCA_LU62_ARM_SUFFIX	MQ_ARM_SUFFIX_LENGTH	Suffix of the SYS1.PARMLIB member APPCPM <i>xx</i> , that nominates the LUADD for this channel initiator	z/OS
MQCA_PARENT	MQ_Q_MGR_NAME_LENGTH	Name of a hierarchically connected queue manager that is nominated as the parent of this queue manager	
MQCA_Q_MGR_DESC	MQ_Q_MGR_DESC_LENGTH	Queue manager description	
MQCA_Q_MGR_IDENTIFIER	MQ_Q_MGR_IDENTIFIER_LENGTH	Queue-manager identifier (H)	
MQCA_Q_MGR_NAME	MQ_Q_MGR_NAME_LENGTH	Name of local queue manager	
MQCA_QSG_NAME	MQ_QSG_NAME_LENGTH	Queue-sharing group name	z/OS
MQCA_REPOSITORY_NAME	MQ_CLUSTER_NAME_LENGTH	Name of cluster for which queue manager provides repository services	
MQCA_REPOSITORY_NAMELIST	MQ_NAMELIST_NAME_LENGTH	Name of namelist object containing names of clusters for which queue manager provides repository services	
MQCA_TCP_NAME	MQ_TCP_NAME_LENGTH	Name of the TCP/IP system that you are using	z/OS

Table 558. MQINQ attribute selectors for the queue manager (continued)

Selector	Length of field	Description	Note
MQIA_ACCOUNTING_CONN_OVERRIDE	MQLONG	Override accounting settings	Not z/OS
MQIA_ACCOUNTING_INTERVAL	MQLONG	How often to write intermediate accounting records	Not z/OS
MQIA_ACCOUNTING_MQI	MQLONG	Controls collection of accounting information for MQI data	Not z/OS
MQIA_ACCOUNTING_Q	MQLONG	Controls collection of accounting information for queues	Not z/OS
MQIA_ACTIVE_CHANNELS	MQLONG	Maximum number of channels that can be active at any time	z/OS
MQIA_ADOPTNEWMCA_CHECK	MQLONG	Elements that are checked to determine whether to adopt an MCA. The check is performed when a new inbound channel is detected that has the same name as an MCA that is already active.	z/OS
MQIA_ADOPTNEWMCA_INTERVAL	MQLONG	Amount of time, in seconds, that the new channel waits for the orphaned channel to end	Not z/OS
MQIA_ADOPTNEWMCA_TYPE	MQLONG	Whether to restart an orphaned instance of an MCA of a particular channel type automatically when a new inbound channel request matching the AdoptNewMCACheck parameters is detected	z/OS
MQIA_AUTHORITY_EVENT	MQLONG	Control attribute for authority events	Not z/OS
MQIA_BRIDGE_EVENT	MQLONG	Control attribute for IMS bridge events	z/OS
MQIA_CHANNEL_AUTO_DEF	MQLONG	Control attribute for automatic channel definition	Not z/OS
MQIA_CHANNEL_AUTO_DEF_EVENT	MQLONG	Control attribute for automatic channel definition events	Not z/OS
MQIA_CHANNEL_EVENT	MQLONG	Control attribute for channel events	
MQIA_CHINIT_ADAPTERS	MQLONG	Number of adapter subtasks to use for processing IBM MQ calls	z/OS
MQIA_CHINIT_DISPATCHERS	MQLONG	Number of dispatchers to use for the channel initiator	z/OS
MQIA_CHINIT_TRACE_AUTOSTART	MQLONG	Whether to start channel initiator trace automatically	z/OS
MQIA_CHINIT_TRACE_TABLE_SIZE	MQLONG	Size of the trace data space (in MB) of the channel initiator	z/OS
MQIA_CLUSTER_WORKLOAD_LENGTH	MQLONG	Cluster workload length.	

Table 558. MQINQ attribute selectors for the queue manager (continued)

Selector	Length of field	Description	Note
MQIA_CLWL_MRU_CHANNELS	MQLONG	Number of most recently used channels for cluster workload balancing	
MQIA_CLWL_USEQ	MQLONG	Use remote queues	
MQIA_CODED_CHAR_SET_ID	MQLONG	Coded character set identifier	
MQIA_COMMAND_EVENT	MQLONG	Control attribute for command events	
MQIA_COMMAND_LEVEL	MQLONG	Command level supported by queue manager	
MQIA_CONFIGURATION_EVENT	MQLONG	Control attribute for configuration events	Not z/OS
MQIA_DEF_CLUSTER_XMIT_Q_TYPE	MQLONG	Default transmission queue type to be used for cluster-sender channels.	
MQIA_DIST_LISTS	MQLONG	Distribution list support	Not z/OS
MQIA_DNS_WLM	MQLONG	Whether the TCP listener that handles inbound transmissions for the queue-sharing group registers with Workload Manager for Dynamic Domain Name Services	z/OS
MQIA_EXPIRY_INTERVAL	MQLONG	Interval between scans for expired messages	z/OS
MQIA_GROUP_UR	MQLONG	Control attribute for whether GROUP units of recovery are enabled for this queue manager. The GROUP unit of recovery disposition is only available if the queue manager is a member of a queue-sharing group	z/OS
MQIA_IGQ_PUT_AUTHORITY	MQLONG	Intra-group queuing put authority	z/OS
MQIA_INHIBIT_EVENT	MQLONG	Control attribute for inhibit events	Not z/OS
MQIA_INTRA_GROUP_queuing	MQLONG	Intra-group queuing support	z/OS
MQIA_LISTENER_TIMER	MQLONG	Time interval (in seconds) between IBM MQ attempts to restart the listener if APPC or TCP/IP failed.	z/OS
MQIA_LOCAL_EVENT	MQLONG	Control attribute for local events	Not z/OS
MQIA_LOGGER_EVENT	MQLONG	Control attribute for inhibit events	Not z/OS
MQIA_LU62_CHANNELS	MQLONG	Maximum number of channels that can be current, or clients that can be connected, using the LU 6.2 transmission protocol	z/OS

Table 558. MQINQ attribute selectors for the queue manager (continued)


Selector	Length of field	Description	Note
MQIA_MSG_MARK_BROWSE_INTERVAL	MQLONG	Time interval (in milliseconds) after which the queue manager can automatically remove a mark from browse messages.  Attention: You should not set this value below the default of 5000.	
MQIA_MAX_CHANNELS	MQLONG	Maximum number of channels that can be current (including server-connection channels with connected clients)	z/OS
MQIA_MAX_HANDLES	MQLONG	Maximum number of handles	
MQIA_MAX_MSG_LENGTH	MQLONG	Maximum message length	
MQIA_MAX_PRIORITY	MQLONG	Maximum priority	
MQIA_MAX_UNCOMMITTED_MESSAGES	MQLONG	Maximum number of uncommitted messages within a unit of work	
MQIA_OUTBOUND_PORT_MAX	MQLONG	With MQIA_OUTBOUND_PORT_MIN, defines range of port numbers to use when binding outgoing channels	z/OS
MQIA_OUTBOUND_PORT_MIN	MQLONG	With MQIA_OUTBOUND_PORT_MAX, defines range of port numbers to use when binding outgoing channels	z/OS
MQIA_PERFORMANCE_EVENT	MQLONG	Control attribute for performance events	Not z/OS
MQIA_PLATFORM	MQLONG	Platform on which the queue manager resides	
MQIA_PROT_POLICY_CAPABILITY	MQLONG	Indicates whether security capabilities of IBM MQ Advanced Message Security are available for a queue manager.	
MQIA_PUBSUB_MAXMSG_RETRY_COUNT	MQLONG	The number of attempts to reprocess a failed command message under sync point	
MQIA_PUBSUB_MODE	MQLONG	Whether the publish/subscribe engine and the queued publish/subscribe interface are running. Applications to publish or subscribe using the application programming interface require the publish/subscribe engine. Queues that are monitored by the queued publish/subscribe interface require the queued publish/subscribe interface to be running.	
MQIA_PUBSUB_NP_MSG	MQLONG	Whether to discard (or keep) an undelivered input message	

Table 558. MQINQ attribute selectors for the queue manager (continued)

Selector	Length of field	Description	Note
MQIA_PUBSUB_NP_RESP	MQLONG	Controls the behavior of undelivered response messages	
MQIA_PUBSUB_SYNC_PT	MQLONG	Whether only persistent (or all) messages are processed under sync point	
MQIA_QMGR_CFCONLOS	MQLONG	Specifies the action to be taken when the queue manager loses connectivity to the administration structure or any CF structures with CFCONLOS set to ASQMGR	z/OS
MQIA_RECEIVE_TIMEOUT	MQLONG	Approximately how long a TCP/IP channel waits to receive data, including heartbeats, from its partner, before returning to the inactive state. The value is numeric, qualified by MQIA_RECEIVE_TIMEOUT_TYPE.	z/OS
MQIA_RECEIVE_TIMEOUT_MIN	MQLONG	Minimum time that a TCP/IP channel waits to receive data, including heartbeats, from its partner, before returning to the inactive state	z/OS
MQIA_RECEIVE_TIMEOUT_TYPE	MQLONG	Approximately how long a TCP/IP channel waits to receive data, including heartbeats, from its partner, before returning to the inactive state. MQIA_RECEIVE_TIMEOUT_TYPE is the qualifier applied to MQIA_RECEIVE_TIMEOUT.	z/OS
MQIA_REMOTE_EVENT	MQLONG	Control attribute for remote events	Not z/OS
MQIA_SECURITY_CASE	MQLONG	Case of security profiles	z/OS
MQIA_SSL_EVENT	MQLONG	Control attribute for channel events	
MQIA_SSL_FIPS_REQUIRED	MQLONG	Use only FIPS-certified algorithms for cryptography	
MQIA_SSL_RESET_COUNT	MQLONG	SSL key reset count	
MQIA_START_STOP_EVENT	MQLONG	Control attribute for start stop events	Not z/OS
MQIA_STATISTICS_AUTO_CLUSTER	MQLONG	Controls collection of statistics monitoring information for cluster sender channels	
MQIA_STATISTICS_CHANNEL	MQLONG	Controls collection of statistics data for channels	
MQIA_STATISTICS_INTERVAL	MQLONG	How often to write statistics monitoring data	Not z/OS

Table 558. MQINQ attribute selectors for the queue manager (continued)			
Selector	Length of field	Description	Note
MQIA_STATISTICS_MQI	MQLONG	Controls collection of statistics monitoring information for queue manager	Not z/OS
MQIA_STATISTICS_Q	MQLONG	Controls collection of statistics data for queues	Not z/OS
MQIA_SYNCPOINT	MQLONG	sync point availability	
MQIA_TCP_CHANNELS	MQLONG	Maximum number of channels that can be current, or clients that can be connected, using the TCP/IP transmission protocol	z/OS
MQIA_TCP_KEEP_ALIVE	MQLONG	Whether to use the TCP KEEPALIVE facility to check that the other end of the connection is still available	z/OS
MQIA_TCP_STACK_TYPE	MQLONG	Whether the channel initiator can use only the TCP/IP address space specified in TCPNAME, or can optionally bind to any selected TCP/IP address	z/OS
MQIA_TRACE_ROUTE_RECORDING	MQLONG	Controls recording of trace-route information	z/OS
MQIA_TREE_LIFE_TIME	MQLONG	Lifetime of unused non-administrative topics	
MQIA_TRIGGER_INTERVAL	MQLONG	Trigger interval	

IntAttrCount

Type: MQLONG - input

This is the number of elements in the *IntAttrs* array. Zero is a valid value.

If *IntAttrCount* is at least the number of MQIA_* selectors in the *Selectors* parameter, all integer attributes requested are returned.

IntAttrs

Type: MQLONG x *IntAttrCount* - output

This is an array of *IntAttrCount* integer attribute values.

Integer attribute values are returned in the same order as the MQIA_* selectors in the *Selectors* parameter. If the array contains more elements than the number of MQIA_* selectors, the excess elements are unchanged.

If *Hobj* represents a queue, but an attribute selector does not apply to that type of queue, the specific value MQIAV_NOT_APPLICABLE is returned. It is returned for the corresponding element in the *IntAttrs* array.

If the *IntAttrCount* or *SelectorCount* parameter is zero, *IntAttrs* is not referred to. In this case, the parameter address passed by programs written in C or S/390 assembler might be null.

CharAttrLength

Type: MQLONG - input

This is the length in bytes of the *CharAttrs* parameter.

`CharAttrLength` must be at least the sum of the lengths of the requested character attributes (see [Selectors](#)). Zero is a valid value.

CharAttrs

Type: `MQCHAR` x `CharAttrLength` - output

This is the buffer in which the character attributes are returned, concatenated together. The length of the buffer is given by the `CharAttrLength` parameter.

Character attributes are returned in the same order as the `MQCA_*` selectors in the `Selectors` parameter. The length of each attribute string is fixed for each attribute (see [Selectors](#)), and the value in it is padded to the right with blanks if necessary. You can provide a buffer larger than needed to contain all the requested character attributes and padding. The bytes beyond the last attribute value returned are unchanged.

If *Hobj* represents a queue, but an attribute selector does not apply to that type of queue, a character string consisting entirely of asterisks (*) is returned. The asterisk is returned as the value of that attribute in `CharAttrs`.

If the `CharAttrLength` or `SelectorCount` parameter is zero, `CharAttrs` is not referred to. In this case, the parameter address passed by programs written in C or S/390 assembler might be null.

CompCode

Type: `MQLONG` - output

The completion code:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: `MQLONG` - output

If `CompCode` is `MQCC_OK`:

MQRC_NONE

(0, X'000') No reason to report.

If `CompCode` is `MQCC_WARNING`:

MQRC_CHAR_ATTRS_TOO_SHORT

(2008, X'7D8') Not enough space allowed for character attributes.

MQRC_INT_ATTR_COUNT_TOO_SMALL

(2022, X'7E6') Not enough space allowed for integer attributes.

MQRC_SELECTOR_NOT_FOR_TYPE

(2068, X'814') Selector not applicable to queue type.

If `CompCode` is `MQCC_FAILED`:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS
 (2219, X'8AB ') MQI call entered before previous call complete.

MQRC_CF_STRUC_FAILED
 (2373, X'945 ') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE
 (2346, X'92A ') Coupling-facility structure in use.

MQRC_CHAR_ATTR_LENGTH_ERROR
 (2006, X'7D6 ') Length of character attributes not valid.

MQRC_CHAR_ATTRS_ERROR
 (2007, X'7D7 ') Character attributes string not valid.

MQRC_CICS_WAIT_FAILED
 (2140, X'85C ') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN
 (2009, X'7D9 ') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED
 (2217, X'8A9 ') Not authorized for connection.

MQRC_CONNECTION_STOPPING
 (2203, X'89B ') Connection shutting down.

MQRC_HCONN_ERROR
 (2018, X'7E2 ') Connection handle not valid.

MQRC_HOBJ_ERROR
 (2019, X'7E3 ') Object handle not valid.

MQRC_INT_ATTR_COUNT_ERROR
 (2021, X'7E5 ') Count of integer attributes not valid.

MQRC_INT_ATTRS_ARRAY_ERROR
 (2023, X'7E7 ') Integer attributes array not valid.

MQRC_NOT_OPEN_FOR_INQUIRE
 (2038, X'7F6 ') Queue not open for inquire.

MQRC_OBJECT_CHANGED
 (2041, X'7F9 ') Object definition changed since opened.

MQRC_OBJECT_DAMAGED
 (2101, X'835 ') Object damaged.

MQRC_PAGESET_ERROR
 (2193, X'891 ') Error accessing page-set data set.

MQRC_Q_DELETED
 (2052, X'804 ') Queue deleted.

MQRC_Q_MGR_NAME_ERROR
 (2058, X'80A ') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE
 (2059, X'80B ') Queue manager not available for connection.

MQRC_Q_MGR_STOPPING
 (2162, X'872 ') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM
 (2102, X'836 ') Insufficient system resources available.

MQRC_SELECTOR_COUNT_ERROR
 (2065, X'811 ') Count of selectors not valid.

MQRC_SELECTOR_ERROR
 (2067, X'813 ') Attribute selector not valid.

MQRC_SELECTOR_LIMIT_EXCEEDED
 (2066, X'812 ') Count of selectors too large.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes; see [Messages and reason codes](#)

Usage notes

1. The values returned are a snapshot of the selected attributes. There is no guarantee that the attributes remain the same before the application can act upon the returned values.
2. When you open a model queue, a dynamic local queue is created. A dynamic local queue is created even if you open the model queue to inquire about its attributes.

The attributes of the dynamic queue are largely the same as the attributes of the model queue at the time that the dynamic queue is created. If you then use the MQINQ call on this queue, the queue manager returns the attributes of the dynamic queue, and not the attributes of the model queue. See [Table 561 on page 803](#) for details of which attributes of the model queue are inherited by the dynamic queue.

3. If the object being inquired is an alias queue, the attribute values returned by the MQINQ call are the attributes of the alias queue. They are not the attributes of the base queue or topic to which the alias resolves.
4. If the object being inquired is a cluster queue, the attributes that can be inquired depend on how the queue is opened:
 - You can open a cluster queue for inquire plus one or more of the input, browse, or set operations. To do so, there must be a local instance of the cluster queue for the open to succeed. In this case, the attributes that can be inquired are the attributes that are valid for local queues.

If the cluster queue is open for inquire without input, browse, or set specified, the call returns completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE (2068) if you attempt to inquire attributes which are valid only for local queues, and not cluster queues.

- You can open a cluster queue for inquire while passing the base queue manager name of the connected queue manager.

To do so, there must be a local instance of the cluster queue for the open to succeed. If the base queue manager is not passed, the call returns completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE (2068) if you attempt to inquire attributes which are valid only for local queues, and not cluster queues.

- If the cluster queue is opened for inquire alone, or inquire and output, only the attributes listed can be inquired. The **QType** attribute has the value MQQT_CLUSTER in this case:

- MQCA_Q_DESC
- MQCA_Q_NAME
- MQIA_DEF_BIND
- MQIA_DEF_PERSISTENCE
- MQIA_DEF_PRIORITY
- MQIA_INHIBIT_PUT
- MQIA_Q_TYPE

You can open the cluster queue with no fixed binding. You can open it with MQOO_BIND_NOT_FIXED specified on the MQOPEN call. Alternatively, specify MQOO_BIND_AS_Q_DEF, and set the **DefBind** attribute of the queue to MQBND_BIND_NOT_FIXED. If you open a cluster queue with no fixed binding, successive MQINQ calls for the queue might inquire different instances of the cluster queue. However, it is typical for all the instances have the same attribute values.

- An alias queue object can be defined for a cluster. Because TARGTYPE and TARGET are not cluster attributes, the process performing an MQOPEN process on the alias queue is not aware of the object to which the alias resolves.

During the initial MQOPEN, the alias queue resolves to a queue manager and a queue in the cluster. Name resolution takes place again at the remote queue manager and it is here that the TARGTYPE of the alias queue is resolved.

If the alias queue resolves to a topic alias, then publication of messages put to the alias queue takes place at this remote queue manager.

See [Cluster queues](#)

5. You might want to inquire a number of attributes, and then set some of them using the MQSET call. To program inquire and set efficiently, position the attributes to be set at the beginning of the selector arrays. If you do so, the same arrays with reduced counts can be used for MQSET.
6. If more than one of the warning situations arise (see the *CompCode* parameter), the reason code returned is the first one in the following list that applies:
 - a. MQRC_SELECTOR_NOT_FOR_TYPE
 - b. MQRC_INT_ATTR_COUNT_TOO_SMALL
 - c. MQRC_CHAR_ATTRS_TOO_SHORT
7. The following topic have information about object attributes:
 - [“Attributes for queues” on page 801](#)
 - [“Attributes for namelists” on page 832](#)
 - [“Attributes for process definitions” on page 834](#)
 - [“Attributes for the queue manager” on page 768](#)

C invocation

```
MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
      CharAttrLength, CharAttrs, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQHOBJ    Hobj;           /* Object handle */
MQLONG    SelectorCount;  /* Count of selectors */
MQLONG    Selectors[n];   /* Array of attribute selectors */
MQLONG    IntAttrCount;   /* Count of integer attributes */
MQLONG    IntAttrs[n];    /* Array of integer attributes */
MQLONG    CharAttrLength; /* Length of character attributes buffer */
MQCHAR    CharAttrs[n];   /* Character attributes */
MQLONG    CompCode;       /* Completion code */
MQLONG    Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQINQ' USING HCONN, HOBJ, SELECTORCOUNT, SELECTORS-TABLE,
                  INTATTRCOUNT, INTATTRS-TABLE, CHARATTRLENGTH,
                  CHARATTRS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Object handle
01 HOBJ           PIC S9(9) BINARY.
** Count of selectors
01 SELECTORCOUNT PIC S9(9) BINARY.
** Array of attribute selectors
01 SELECTORS-TABLE.
02 SELECTORS      PIC S9(9) BINARY OCCURS n TIMES.
```

```

** Count of integer attributes
01 INTATTRCOUNT PIC S9(9) BINARY.
** Array of integer attributes
01 INTATTRS-TABLE.
02 INTATTRS PIC S9(9) BINARY OCCURS n TIMES.
** Length of character attributes buffer
01 CHARATTRLENGTH PIC S9(9) BINARY.
** Character attributes
01 CHARATTRS PIC X(n).
** Completion code
01 COMPCODE PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON PIC S9(9) BINARY.

```

PL/I invocation

```

call MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,
            IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason);

```

Declare the parameters as follows:

```

dcl Hconn          fixed bin(31); /* Connection handle */
dcl Hobj           fixed bin(31); /* Object handle */
dcl SelectorCount  fixed bin(31); /* Count of selectors */
dcl Selectors(n)   fixed bin(31); /* Array of attribute selectors */
dcl IntAttrCount   fixed bin(31); /* Count of integer attributes */
dcl IntAttrs(n)    fixed bin(31); /* Array of integer attributes */
dcl CharAttrLength fixed bin(31); /* Length of character attributes
                                buffer */
dcl CharAttrs      char(n);       /* Character attributes */
dcl CompCode       fixed bin(31); /* Completion code */
dcl Reason         fixed bin(31); /* Reason code qualifying
                                CompCode */

```

High Level Assembler invocation

```

CALL MQINQ,(HCONN,HOBJ,SELECTORCOUNT,SELECTORS,INTATTRCOUNT, X
            INTATTRS,CHARATTRLENGTH,CHARATTRS,COMPCODE,REASON)

```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HOBJ	DS	F	Object handle
SELECTORCOUNT	DS	F	Count of selectors
SELECTORS	DS	(n)F	Array of attribute selectors
INTATTRCOUNT	DS	F	Count of integer attributes
INTATTRS	DS	(n)F	Array of integer attributes
CHARATTRLENGTH	DS	F	Length of character attributes buffer
CHARATTRS	DS	CL(n)	Character attributes
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

Visual Basic invocation

```

MQINQ Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
CharAttrLength, CharAttrs, CompCode, Reason

```

Declare the parameters as follows:

Dim Hconn	As Long	'Connection handle'
Dim Hobj	As Long	'Object handle'
Dim SelectorCount	As Long	'Count of selectors'
Dim Selectors	As Long	'Array of attribute selectors'
Dim IntAttrCount	As Long	'Count of integer attributes'
Dim IntAttrs	As Long	'Array of integer attributes'
Dim CharAttrLength	As Long	'Length of character attributes buffer'
Dim CharAttrs	As String	'Character attributes'
Dim CompCode	As Long	'Completion code'
Dim Reason	As Long	'Reason code qualifying CompCode'

MQINQMP - Inquire message property

The MQINQMP call returns the value of a property of a message.

Syntax

MQINQMP (*Hconn*, *Hmsg*, *InqPropOpts*, *Name*, *PropDesc*, *Type*, *ValueLength*, *Value*, *DataLength*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using MQHC_UNASSOCIATED_HCONN then a valid connection must be established on the thread inquiring a property of the message handle otherwise the call fails with MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMSG - input

This is the message handle to be inquired. The value was returned by a previous **MQCRTMH** call.

InqPropOpts

Type: MQIMPO - input/output

See the [MQIMPO](#) data type for details.

Name

Type: MQCHARV - input/output

The name of the property to inquire.

If no property with this name can be found, the call fails with reason MQRC_PROPERTY_NOT_AVAILABLE.

You can use the wildcard character percent sign (%) at the end of the property name. The wildcard matches zero or more characters, including the period (.) character. This allows an application to inquire the value of many properties. Call MQINQMP with option MQIMPO_INQ_FIRST to get the first matching property and again with the option MQIMPO_INQ_NEXT to get the next matching property. When no more matching properties are available, the call fails with MQRC_PROPERTY_NOT_AVAILABLE. If the *ReturnedName* field of the InqPropOpts structure is initialized with an address or offset for the returned name of the property, this is completed on return from MQINQMP with the name of the property that has been matched. If the *VSBufSize* field of the *ReturnedName* in the InqPropOpts structure is less than the length of the returned property name the completion code is set MQCC_FAILED with reason MQRC_PROPERTY_NAME_TOO_BIG.

Properties that have known synonyms are returned as follows:

1. Properties with the prefix "mqps." are returned as the IBM MQ property name. For example, "MQTopicString" is the returned name rather than "mqps.Top"
2. Properties with the prefix "jms." or "mcd." are returned as the JMS header field name, for example, "JMSExpiration" is the returned name rather than "jms.Exp".
3. Properties with the prefix "usr." are returned without that prefix, for example, "Color" is returned rather than "usr.Color".

Properties with synonyms are only returned once.

In the C programming language, the following macro variables are defined for inquiring on all properties and then all properties that begin "usr.":

MQPROP_INQUIRE_ALL

Inquire on all properties of the message.

MQPROP_INQUIRE_ALL can be used in the following way:

```
MQCHARV Name = {MQPROP_INQUIRE_ALL};
```

MQPROP_INQUIRE_ALL_USR

Inquire on all properties of the message that start "usr.". The returned name is returned without the "usr." prefix.

If MQIMP_INQ_NEXT is specified but Name has changed since the previous call or this is the first call, then MQIMPO_INQ_FIRST is implied.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

PropDesc

Type: MQPD - output

This structure is used to define the attributes of a property, including what happens if the property is not supported, what message context the property belongs to, and what messages the property should be copied into. See [MQPD](#) for details of this structure.

Type

Type: MQLONG - input/output

On return from the MQINQMP call this parameter is set to the data type of *Value*. The data type can be any of the following:

MQTYPE_BOOLEAN

A boolean.

MQTYPE_BYTE_STRING

a byte string.

MQTYPE_INT8

An 8-bit signed integer.

MQTYPE_INT16

A 16-bit signed integer.

MQTYPE_INT32

A 32-bit signed integer.

MQTYPE_INT64

A 64-bit signed integer.

MQTYPE_FLOAT32

A 32-bit floating-point number.

MQTYPE_FLOAT64

A 64-bit floating-point number.

MQTYPE_STRING

A character string.

MQTYPE_NULL

The property exists but has a null value.

If the data type of the property value is not recognized then MQTYPE_STRING is returned and a string representation of the value is placed into the *Value* area. A string representation of the data type can be found in the *TypeString* field of the *InqPropOpts* parameter. A warning completion code is returned with reason MQRC_PROP_TYPE_NOT_SUPPORTED.

Additionally, if the option MQIMPO_CONVERT_TYPE is specified, conversion of the property value is requested. Use *Type* as an input to specify the data type that you want the property to be returned as. See the description of the [MQIMPO_CONVERT_TYPE](#) option of the [MQIMPO](#) structure for details of data type conversion.

If you do not request type conversion, you can use the following value on input:

MQTYPE_AS_SET

The value of the property is returned without converting its data type.

ValueLength

Type: MQLONG - input

The length in bytes of the Value area. Specify zero for properties that you do not require the value returned for. These could be properties which are designed by an application to have a null value or an empty string. Also specify zero if the [MQIMPO_QUERY_LENGTH](#) option has been specified; in this case no value is returned.

Value

Type: MQBYTE *ValueLength* - output

This is the area to contain the inquired property value. The buffer should be aligned on a boundary appropriate for the value being returned. Failure to do so can result in an error when the value is later accessed.

If *ValueLength* is less than the length of the property value, as much of the property value as possible is moved into *Value* and the call fails with completion code MQCC_FAILED and reason MQRC_PROPERTY_VALUE_TOO_BIG.

The character set of the data in *Value* is given by the ReturnedCCSID field in the InqPropOpts parameter. The encoding of the data in *Value* is given by the ReturnedEncoding field in the InqPropOpts parameter.

In the C programming language, the parameter is declared as a pointer-to-void; the address of any type of data can be specified as the parameter.

If the *ValueLength* parameter is zero, *Value* is not referred to and its value passed by programs written in C or System/390 assembler can be null.

DataLength

Type: MQLONG - output

This is the length in bytes of the actual property value as returned in the *Value* area.

If *DataLength* is less than the property value length, *DataLength* is still filled in on return from the MQINQMP call. This allows the application to determine the size of the buffer required to accommodate the property value, and then reissue the call with a buffer of the appropriate size.

The following values can also be returned.

If the *Type* parameter is set to MQTYPE_STRING or MQTYPE_BYTE_STRING:

MQVL_EMPTY_STRING

The property exists but contains no characters or bytes.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_PROP_NAME_NOT_CONVERTED

(2492, X'09BC') Returned property name not converted.

MQRC_PROP_VALUE_NOT_CONVERTED

(2466, X'09A2') Property value not converted.

MQRC_PROP_TYPE_NOT_SUPPORTED

(2467, X'09A3') Property data type is not supported.

MQRC_RFH_FORMAT_ERROR

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'0852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'086D') Primary and home ASIDs differ.

MQRC_BUFFER_ERROR

(2004, X'07D4') Value parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'07D5') Value length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CONNECTION_BROKEN

(2009, X'07D9') Connection to queue manager lost.

MQRC_DATA_LENGTH_ERROR

(2010, X'07DA') Data length parameter not valid.

MQRC_IMPO_ERROR

(2464, X'09A0') Inquire message property options structure not valid.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07F8') Options not valid or not consistent.

MQRC_PD_ERROR

(2482, X'09B2') Property descriptor structure not valid.

MQRC_PROP_CONV_NOT_SUPPORTED

(2470, X'09A6') Conversion from the actual to requested data type not supported.

MQRC_PROPERTY_NAME_ERROR

(2442, X'098A') Invalid property name.

MQRC_PROPERTY_NAME_TOO_BIG

(2465, X'09A1') Property name too large for returned name buffer.

MQRC_PROPERTY_NOT_AVAILABLE

(2471, X'09A7') Property not available.

MQRC_PROPERTY_VALUE_TOO_BIG

(2469, X'09A5') Property value too large for the Value area.

MQRC_PROP_NUMBER_FORMAT_ERROR

(2472, X'09A8') Number format error encountered in value data.

MQRC_PROPERTY_TYPE_ERROR

(2473, X'09A9') Invalid requested property type.

MQRC_SOURCE_CCSID_ERROR

(2111, X'083F') Property name coded character set identifier not valid.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'0871') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'0893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C invocation

```
MQINQMP (Hconn, Hmsg, &InqPropOpts, &Name, &PropDesc, &Type,
ValueLength, Value, &DataLength, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN Hconn;          /* Connection handle */
MQHMSG  Hmsg;           /* Message handle */
MQIMPO  InqPropOpts;    /* Options that control the action of MQINQMP */
MQCHARV Name;          /* Property name */
MQPD    PropDesc;       /* Property descriptor */
MQLONG  Type;           /* Property data type */
MQLONG  ValueLength;    /* Length in bytes of the Value area */
MQBYTE  Value[n];       /* Area to contain the property value */
MQLONG  DataLength;     /* Length of the property value */
MQLONG  CompCode;       /* Completion code */
MQLONG  Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQINQMP' USING HCONN, HMSG, INQMSGOPTS, NAME, PROPDSC, TYPE,
VALUELENGTH, VALUE, DATALENGTH, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Message handle
01 HMSG           PIC S9(18) BINARY.
** Options that control the action of MQINQMP
01 INQMSGOPTS.
  COPY CMQIMPOV.
** Property name
01 NAME.
  COPY CMQCHRVV.
** Property descriptor
01 PROPDSC.
  COPY CMQPDV.
** Property data type
01 TYPE          PIC S9(9) BINARY.
** Length in bytes of the VALUE area
01 VALUELENGTH PIC S9(9) BINARY.
** Area to contain the property value
01 VALUE         PIC X(n).
** Length of the property value
01 DATALENGTH  PIC S9(9) BINARY.
** Completion code
01 COMPCODE     PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON       PIC S9(9) BINARY.
```

PL/I invocation

```
call MQINQMP (Hconn, Hmsg, InqPropOpts, Name, PropDesc, Type,
ValueLength, Value, DataLength, CompCode, Reason);
```

Declare the parameters as follows:

```
dc1 Hconn      fixed bin(31); /* Connection handle */
dc1 Hmsg       fixed bin(63); /* Message handle */
dc1 InqPropOpts like MQIMPO;  /* Options that control the action of MQINQMP */
dc1 Name       like MQCHARV;  /* Property name */
dc1 PropDesc   like MQPD;     /* Property descriptor */
dc1 Type       fixed bin (31); /* Property data type */
dc1 ValueLength fixed bin (31); /* Length in bytes of the Value area */
dc1 Value      char (n);      /* Area to contain the property value */
dc1 DataLength fixed bin (31); /* Length of the property value */
dc1 CompCode   fixed bin (31); /* Completion code */
dc1 Reason     fixed bin (31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQINQMP, (HCONN,HMSG,INQMSGOPTS,NAME,PROPDSC,TYPE,
VALUELENGTH,VALUE,DATALength,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HMSG	DS	D	Message handle
INQMSGOPTS	CMQIMPOA	,	Options that control the action of MQINQMP
NAME	CMQCHRVA	,	Property name
PROPDSC	CMQPDA	,	Property descriptor
TYPE	DS	F	Property data type
VALUELENGTH	DS	F	Length in bytes of the VALUE area
VALUE	DS	CL(n)	Area to contain the property value
DATALength	DS	F	Length of the property value
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQMHBUF - Convert message handle into buffer

The MQMHBUF call converts a message handle into a buffer and is the inverse of the MQBUFMH call.

Syntax

MQMHBUF (*Hconn*, *Hmsg*, *MsgHBufOpts*, *Name*, *MsgDesc*, *BufferLength*, *Buffer*, *DataLength*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using MQHC_UNASSOCIATED_HCONN, a valid connection must be established on the thread deleting the message handle. If a valid connection is not established, the call fails with MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMSG - input

This is the message handle for which a buffer is required. The value was returned by a previous MQCRTMH call.

MsgHBufOpts

Type: MQMHBO - input

The MQMHBO structure allows applications to specify options that control how buffers are produced from message handles.

See [“MQMHBO - Message handle to buffer options” on page 453](#) for details.

Name

Type: MQCHARV - input

The name of the property or properties to put into the buffer.

If no property matching the name can be found, the call fails with MQRC_PROPERTY_NOT_AVAILABLE.

You can use a wildcard to put more than one property into the buffer. To do this, use the wildcard character '%' at the end of the property name. This wildcard matches zero or more characters, including the '.' character.

In the C programming language, the following macro variables are defined for inquiring on all properties and all properties that begin 'usr':

MQPROP_INQUIRE_ALL

Put all properties of the message into the buffer

MQPROP_INQUIRE_ALL_USR

Put all properties of the message that start with the characters 'usr.' into the buffer.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

MsgDesc

Type: MQMD - input/output

The *MsgDesc* structure describes the contents of the buffer area.

On output, the *Encoding*, *CodedCharSetId* and *Format* fields are set to correctly describe the encoding, character set identifier, and format of the data in the buffer area as written by the call.

Data in this structure is in the character set and encoding of the application.

BufferLength

Type: MQLONG - input

BufferLength is the length of the Buffer area, in bytes.

Buffer

Type: MQBYTEXBufferLength - output

Buffer defines the area to contain the message properties. You must align the buffer on a 4-byte boundary.

If *BufferLength* is less than the length required to store the properties in *Buffer*, MQMHBUF fails with MQRC_PROPERTY_VALUE_TOO_BIG.

The contents of the buffer can change even if the call fails.

DataLength

Type: MQLONG - output

DataLength is the length, in bytes, of the returned properties in the buffer. If the value is zero, no properties matched the value given in *Name* and the call fails with reason code MQRC_PROPERTY_NOT_AVAILABLE.

If *BufferLength* is less than the length required to store the properties in the buffer, the MQMHBUF call fails with MQRC_PROPERTY_VALUE_TOO_BIG, but a value is still entered into *DataLength*. This allows the application to determine the size of the buffer required to accommodate the properties, and then reissue the call with the required *BufferLength*.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_MHBO_ERROR

(2501, X'095C') Message handle to buffer options structure not valid.

MQRC_BUFFER_ERROR

(2004, X'07D4') Buffer parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'07D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_CONNECTION_BROKEN

(2009, X'07D9') Connection to queue manager lost.

MQRC_DATA_LENGTH_ERROR

(2010, X'07DA') Data length parameter not valid.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle not valid.

MQRC_MD_ERROR

(2026, X'07EA') Message descriptor not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_PROPERTY_NAME_ERROR

(2442, X'098A') Property name is not valid.

MQRC_PROPERTY_NOT_AVAILABLE

(2471, X'09A7') Property not available.

MQRC_PROPERTY_VALUE_TOO_BIG

(2469, X'09A5') BufferLength value is too small to contain specified properties.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C invocation

```
MQMHBUF (Hconn, Hmsg, &MsgHBufOpts, &Name, &MsgDesc, BufferLength, Buffer,  
         &DataLength, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN Hconn;          /* Connection handle */  
MQHMSG  Hmsg;           /* Message handle */  
MQMHBO  MsgHBufOpts;    /* Options that control the action of MQMHBUF */  
MQCHARV Name;           /* Property name */  
MQMD     MsgDesc;       /* Message descriptor */  
MQLONG  BufferLength;    /* Length in bytes of the Buffer area */  
MQBYTE  Buffer[n];       /* Area to contain the properties */  
MQLONG  DataLength;     /* Length of the properties */  
MQLONG  CompCode;       /* Completion code */  
MQLONG  Reason;         /* Reason code qualifying CompCode */
```

Usage notes

MQMHBUF converts a message handle into a buffer.

You can use it with an MQGET API exit to access certain properties, using the message property APIs, and then pass these in a buffer back to an application designed to use MQRFH2 headers rather than message handles.

This call is the inverse of the MQBUFMH call, which you can use to parse message properties from a buffer into a message handle.

COBOL invocation

```
CALL 'MQMHBUF' USING HCONN, HMSG, MSGHBUFOPTS, NAME, MSGDESC,  
                     BUFFERLENGTH, BUFFER, DATALENGTH, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN          PIC S9(9) BINARY.  
** Message handle  
01 HMSG           PIC S9(18) BINARY.  
** Options that control the action of MQMHBUF  
01 MSGHBUFOPTS.  
   COPY CMQMHBV.  
** Property name  
01 NAME           PIC S9(18) BINARY.  
   COPY CMQCHRVV.  
** Message descriptor  
01 MSGDESC        PIC S9(18) BINARY.  
   COPY CMQMDV.  
** Length in bytes of the Buffer area */  
01 BUFFERLENGTH  PIC S9(9) BINARY.  
** Area to contain the properties  
01 BUFFER         PIC X(n).  
** Length of the properties  
01 DATALENGTH   PIC S9(9) BINARY.  
** Completion code  
01 COMPCODE       PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON         PIC S9(9) BINARY.
```

PL/I invocation

```
call MQMHBUF (Hconn, Hmsg, MsgHBufOpts, Name, MsgDesc, BufferLength, Buffer,  
             DataLength, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn          fixed bin(31); /* Connection handle */  
dcl Hmsg           fixed bin(63); /* Message handle */
```

```

dcl MsgHBufOpts like MQMHBO; /* Options that control the action of MQMHBUF */
dcl Name        like MQCHARV; /* Property name */
dcl MsgDesc     like MQMD;    /* Message descriptor */
dcl BufferLength fixed bin(31); /* Length in bytes of the Buffer area */
dcl Buffer       char(n);      /* Area to contain the properties */
dcl DataLength  fixed bin(31); /* Length of the properties */
dcl CompCode    fixed bin(31); /* Completion code */
dcl Reason      fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```
CALL MQMHBUF, (HCONN,HMSG,MSGHBUFOPTS,NAME,MSGDESC,BUFFERLENGTH,
              BUFFER,DATALLENGTH,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HMSG	DS	D	Message handle
MSGHBUFOPTS	CMQMHB		Options that control the action of MQMHBUF
NAME	CMQCH		Property name
MSGDESC	CMQMD		Message descriptor
BUFFERLENGTH	DS	F	Length in bytes of the BUFFER area
BUFFER	DS	CL(n)	Area to contain the properties
DATALLENGTH	DS	F	Length of the properties
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQOPEN - Open object

The MQOPEN call establishes access to an object.

The following types of object are valid:

- Queue (including distribution lists)
- Namelist
- Process definition
- Queue manager
- Topic

Syntax

MQOPEN (*Hconn*, *ObjDesc*, *Options*, *Hobj*, *CompCode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

ObjDesc

Type: MQOD - input/output

This is a structure that identifies the object to be opened; see “MQOD - Object descriptor” on page 455 for details.

If the *ObjectName* field in the *ObjDesc* parameter is the name of a model queue, a dynamic local queue is created with the attributes of the model queue; this happens whatever options you specify on the *Options* parameter. Subsequent operations using the *Hobj* returned by the MQOPEN call are performed on the new dynamic queue, and not on the model queue. This is true even for the MQINQ and MQSET calls. The name of the model queue in the *ObjDesc* parameter is replaced with the name of the dynamic queue created. The type of the dynamic queue is determined by the value of the *DefinitionType* attribute of the model queue (see “Attributes for queues” on page 801). For information about the close options applicable to dynamic queues, see the description of the MQCLOSE call.

Options

Type: MQLONG - input

You must specify at least one of the following options:

- MQOO_BROWSE
- MQOO_INPUT_* (only one of these)
- MQOO_INQUIRE
- MQOO_OUTPUT
- MQOO_SET
- MQOO_BIND_* (only one of these)

See the following table for details of these options; other options can be specified as required. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations). Combinations that are not valid are noted; all other combinations are valid. Only options that are applicable to the type of object specified by *ObjDesc* are allowed. The following table shows valid MQOPEN options for queries and topics.

Option	Alias ¹	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
<u>MQOO_INPUT_AS_Q_DEF</u>	Yes	Yes	No	No	No	No
<u>MQOO_INPUT_SHARED</u>	Yes	Yes	No	No	No	No
<u>MQOO_INPUT_EXCLUSIVE</u>	Yes	Yes	No	No	No	No
<u>MQOO_OUTPUT</u>	Yes	Yes	Yes	Yes	Yes	Yes
<u>MQOO_BROWSE</u>	Yes	Yes	No	No	No	No
<u>MQOO_CO_OP</u>	Yes	Yes	No	No	No	No
<u>MQOO_INQUIRE</u>	Yes	Yes	<u>2</u>	Yes	No	No
<u>MQOO_SET</u>	Yes	Yes	<u>2</u>	No	No	No
<u>MQOO_BIND_ON_OPEN</u> ³	Yes	Yes	Yes	Yes	Yes	No
<u>MQOO_BIND_NOT_FIXED</u> ³	Yes	Yes	Yes	Yes	Yes	No
<u>MQOO_BIND_ON_GROUP</u> ³	Yes	Yes	Yes	Yes	Yes	No
<u>MQOO_BIND_AS_Q_DEF</u> ³	Yes	Yes	Yes	Yes	Yes	No
<u>MQOO_SAVE_ALL_CONTEXT</u>	Yes	Yes	No	No	No	No
<u>MQOO_PASS_IDENTITY_CONTEXT</u>	Yes	Yes	Yes	Yes	Yes	<u>4</u>
<u>MQOO_PASS_ALL_CONTEXT</u>	Yes	Yes	Yes	Yes	Yes	Yes
<u>MQOO_SET_IDENTITY_CONTEXT</u>	Yes	Yes	Yes	Yes	Yes	<u>4</u>
<u>MQOO_SET_ALL_CONTEXT</u>	Yes	Yes	Yes	Yes	Yes	Yes
<u>MQOO_NO_READ_AHEAD</u>	Yes	Yes	No	No	No	No

Option	Alias ¹	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
MQOO_READ_AHEAD	Yes	Yes	No	No	No	No
MQOO_READ_AHEAD_AS_Q_DEF	Yes	Yes	No	No	No	No
MQOO_ALTERNATE_USER_AUTHORITY	Yes	Yes	Yes	Yes	Yes	Yes
MQOO_FAIL_IF_QUIESCING	Yes	Yes	Yes	Yes	Yes	Yes
MQOO_RESOLVE_LOCAL_Q	Yes	Yes	Yes	Yes	No	No
MQOO_RESOLVE_LOCAL_TOPIC	No	No	No	No	No	Yes
MQOO_NO_MULTICAST	No	No	No	No	No	Yes

Notes:

1. The validity of options for aliases depends on the validity of the option for the queue to which the alias resolves.
2. This option is valid only for the local definition of a remote queue.
3. This option can be specified for any queue type, but is ignored if the queue is not a cluster queue. However, the *DefBind* queue attribute overrides the base queue even when the alias queue is not in a cluster.
4. These attributes can be used with a topic, but affect only the context set for the retained message, not the context fields sent to any subscriber.

Access options: The following options control the type of operations that can be performed on the object:

MQOO_INPUT_AS_Q_DEF

Open queue to get messages using queue-defined default.

The queue is opened for use with subsequent MQGET calls. The type of access is either shared or exclusive, depending on the value of the *DefInputOpenOption* queue attribute; see [“Attributes for queues”](#) on page 801 for details.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_INPUT_SHARED

Open queue to get messages with shared access.

The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with MQOO_INPUT_SHARED, but fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open with MQOO_INPUT_EXCLUSIVE.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_INPUT_EXCLUSIVE

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open by this or another application for input of any type (MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE).

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_OUTPUT

Open queue to put messages, or a topic or topic string to publish messages.

The queue or topic is opened for use with subsequent MQPUT calls.

An MQOPEN call with this option can succeed even if the *InhibitPut* queue attribute is set to MQQA_PUT_INHIBITED (although subsequent MQPUT calls fail while the attribute is set to this value).

This option is valid for all types of queue, including distribution lists, and topics.

The following notes apply to these options:

- Only one of these options can be specified.
- An MQOPEN call with one of these options can succeed even if the *InhibitGet* queue attribute is set to MQQA_GET_INHIBITED (although subsequent MQGET calls fail while the attribute is set to this value).
- If the queue is defined as not being shareable (that is, the *Shareability* queue attribute has the value MQQA_NOT_SHAREABLE), attempts to open the queue for shared access are treated as attempts to open the queue with exclusive access.
- If an alias queue is opened with one of these options, the test for exclusive use (or for whether another application has exclusive use) is against the base queue to which the alias resolves.
- These options are not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_BROWSE

Open queue to browse messages.

The queue is opened for use with subsequent MQGET calls with one of the following options:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_NEXT
- MQGMO_BROWSE_MSG_UNDER_CURSOR

This is allowed even if the queue is currently open for MQOO_INPUT_EXCLUSIVE. An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, and positions it logically before the first message on the queue; see [MQGMO - Options field](#) for further information.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues. It is also not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_CO_OP

Open as a cooperating member of the set of handles.

This option is valid only with the MQOO_BROWSE option. If it is specified without MQOO_BROWSE, MQOPEN returns with MQRC_OPTIONS_ERROR.

The handle returned is considered to be a member of a cooperating set of handles for subsequent MQGET calls with one of the following options:

- MQGMO_MARK_BROWSE_CO_OP
- MQGMO_UNMARKED_BROWSE_MSG
- MQGMO_UNMARK_BROWSE_CO_OP

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_INQUIRE

Open object to inquire attributes.

The queue, namelist, process definition, or queue manager is opened for use with subsequent MQINQ calls.

This option is valid for all types of object other than distribution lists. It is not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_SET

Open queue to set attributes.

The queue is opened for use with subsequent MQSET calls.

This option is valid for all types of queue other than distribution lists. It is not valid if *ObjectQMgrName* is the name of a local definition of a remote queue; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

Binding options: The following options apply when the object being opened is a cluster queue; these options control the binding of the queue handle to an instance of the cluster queue:

MQOO_BIND_ON_OPEN

The local queue manager binds the queue handle to an instance of the destination queue when the queue is opened. As a result, all messages put using this handle are sent to the same instance of the destination queue, and by the same route.

This option is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

MQOO_BIND_NOT_FIXED

This stops the local queue manager binding the queue handle to an instance of the destination queue. As a result, successive MQPUT calls using this handle send the messages to *different* instances of the destination queue, or to the same instance but by different routes. It also allows the instance selected to be changed later by the local queue manager, by a remote queue manager, or by a message channel agent (MCA), according to network conditions.

Note: Client and server applications that need to exchange a *series* of messages to complete a transaction must not use MQOO_BIND_NOT_FIXED (or MQOO_BIND_AS_Q_DEF when *DefBind* has the value MQBND_BIND_NOT_FIXED), because successive messages in the series might be sent to different instances of the server application.

If MQOO_BROWSE or one of the MQOO_INPUT_* options is specified for a cluster queue, the queue manager is forced to select the local instance of the cluster queue. As a result, the binding of the queue handle is fixed, even if MQOO_BIND_NOT_FIXED is specified.

If MQOO_INQUIRE is specified with MQOO_BIND_NOT_FIXED, successive MQINQ calls using that handle might inquire different instances of the cluster queue, although typically all the instances have the same attribute values.

MQOO_BIND_NOT_FIXED is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

MQOO_BIND_ON_GROUP

Allows an application to request that a group of messages are all allocated to the same destination instance.

This option is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

MQOO_BIND_AS_Q_DEF

The local queue manager binds the queue handle in the way defined by the *DefBind* queue attribute. The value of this attribute is either MQBND_BIND_ON_OPEN, MQBND_BIND_NOT_FIXED, or MQBND_BIND_ON_GROUP.

MQOO_BIND_AS_Q_DEF is the default when MQOO_BIND_ON_OPEN, MQOO_BIND_NOT_FIXED, or MQOO_BIND_ON_GROUP is not specified.

MQOO_BIND_AS_Q_DEF aids program documentation. It is not intended that this option is used with either of the other two bind options, but because its value is zero such use cannot be detected.

Context options: The following options control the processing of message context:

MQOO_SAVE_ALL_CONTEXT

Context information is associated with this queue handle. This information is set from the context of any message retrieved using this handle. For more information about message context, see [Message context](#) and [Controlling context information](#).

This context information can be passed to a message that is then put on a queue using the MQPUT or MQPUT1 calls. See the MQPMO_PASS_IDENTITY_CONTEXT and MQPMO_PASS_ALL_CONTEXT options described in “MQPMO - Put-message options” on page 475.

Until a message has been successfully retrieved, context cannot be passed to a message being put on a queue.

A message retrieved using one of the MQGMO_BROWSE_* browse options does not have its context information saved (although the context fields in the *MsgDesc* parameter are set after a browse).

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues. One of the MQOO_INPUT_* options must be specified.

MQOO_PASS_IDENTITY_CONTEXT

This allows the MQPMO_PASS_IDENTITY_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity context information from an input queue that was opened with the MQOO_SAVE_ALL_CONTEXT option. For more information about message context, see [Message context](#) and [Controlling context information](#).

The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

MQOO_PASS_ALL_CONTEXT

This allows the MQPMO_PASS_ALL_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity and origin context information from an input queue that was opened with the MQOO_SAVE_ALL_CONTEXT option. For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies MQOO_PASS_IDENTITY_CONTEXT, which need not therefore be specified. The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

MQOO_SET_IDENTITY_CONTEXT

This allows the MQPMO_SET_IDENTITY_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity context information contained in the *MsgDesc* parameter specified on the MQPUT or MQPUT1 call. For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies MQOO_PASS_IDENTITY_CONTEXT, which need not therefore be specified. The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

MQOO_SET_ALL_CONTEXT

This allows the MQPMO_SET_ALL_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity and origin context information contained in the *MsgDesc* parameter specified on the MQPUT or MQPUT1 call.

For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies the following options, which need not therefore be specified:

- MQOO_PASS_IDENTITY_CONTEXT
- MQOO_PASS_ALL_CONTEXT
- MQOO_SET_IDENTITY_CONTEXT

The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

Read ahead options:

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- Both the client and remote queue manager must be at WebSphere MQ Version 7 or later.
- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.
- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

The following options control whether non-persistent messages are sent to the client before an application requests them. The following notes apply to the read ahead options:

- Only one of these options can be specified.
- These options are valid only for local, alias, and model queues. They are not valid for remote queues, distribution lists, topics or queue managers.
- These options are only applicable when one of MQOO_BROWSE, MQOO_INPUT_SHARED and MQOO_INPUT_EXCLUSIVE are also specified although it is not an error to specify these options with MQOO_INQUIRE or MQOO_SET.
- If the application is not running as an IBM MQ client, these options are ignored.

MQOO_NO_READ_AHEAD

Non-persistent messages are not sent the client before an application requests them.

MQOO_READ_AHEAD

Non-persistent messages are sent to the client before an application requests them.

MQOO_READ_AHEAD_AS_Q_DEF

Read ahead behavior is determined by the default read ahead attribute of the queue being opened. This is the default value.

Other options: The following options control authorization checking, what happens when the queue manager is quiescing, whether to resolve the local queue name, and multicast:

MQOO_ALTERNATE_USER_AUTHORITY

The *AlternateUserId* field in the *ObjDesc* parameter contains a user identifier to use to validate this MQOPEN call. The call can succeed only if this *AlternateUserId* is authorized to open the object with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so. This does not apply to any context options specified, however, which are always checked against the user identifier under which the application is running.

This option is valid for all types of object.

MQOO_FAIL_IF QUIESCING

The MQOPEN call fails if the queue manager is in quiescing state.

On z/OS, for a CICS or IMS application, this option also forces the MQOPEN call to fail if the connection is in quiescing state.

This option is valid for all types of object.

For information about client channels see [Overview of IBM MQ MQI clients](#).

MQOO_RESOLVE_LOCAL_Q

Fill the ResolvedQName in the MQOD structure with the name of the local queue that was opened. Similarly, the ResolvedQMgrName is filled with the name of the local queue manager hosting the local queue. If the MQOD structure is less than Version 3, MQOO_RESOLVE_LOCAL_Q is ignored with no error being returned.

The local queue is always returned when either a local, alias, or model queue is opened, but this is not the case when, for example, a remote queue or a non-local cluster queue is opened without the MQOO_RESOLVE_LOCAL_Q option; the ResolvedQName and ResolvedQMgrName are filled with the RemoteQName and RemoteQMgrName found in the remote queue definition, or similarly with the chosen remote cluster queue.

If you specify MQOO_RESOLVE_LOCAL_Q when opening, for example, a remote queue, ResolvedQName is the transmission queue to which messages are put. The ResolvedQMgrName is filled with the name of the local queue manager hosting the transmission queue.

If you are authorized for browse, input, or output on a queue, you have the required authority to specify this flag on the MQOPEN call. No special authority is needed.

This option is valid only for queues and queue managers.

MQOO_RESOLVE_LOCAL_TOPIC

Fill the ResolvedQName in the MQOD structure with the name of the administrative topic opened.

MQOO_NO_MULTICAST

Publication messages are not sent using multicast.

This option is valid only with the MQOO_OUTPUT option. If it is specified without MQOO_OUTPUT, MQOPEN returns with MQRC_OPTIONS_ERROR.

This option is valid only for a topic.

Hobj

Type: MQHOBJ - output

This handle represents the access that has been established to the object. It must be specified on subsequent IBM MQ calls that operate on the object. It ceases to be valid when the MQCLOSE call is issued, or when the unit of processing that defines the scope of the handle terminates.

The scope of the object handle returned is the same as the scope of the connection handle specified on the call. See [MQCONN - Hconn parameter](#) for information about handle scope.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_MULTIPLE_REASONS

(2136, X'858') Multiple reason codes returned.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ALIAS_BASE_Q_TYPE_ERROR

(2001, X'7D1') Alias base queue not a valid type.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') Coupling facility not available.

MQRC_CF_STRUC_AUTH_FAILED

(2348, X'92C') Coupling-facility structure authorization check failed.

MQRC_CF_STRUC_ERROR

(2349, X'92D') Coupling-facility structure not valid.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CF_STRUC_LIST_HDR_IN_USE

(2347, X'92B') Coupling-facility structure list-header in use.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CLUSTER_EXIT_ERROR

(2266, X'8DA') Cluster workload exit failed.

MQRC_CLUSTER_PUT_INHIBITED

(2268, X'8DC') Put calls inhibited for all queues in cluster.

MQRC_CLUSTER_RESOLUTION_ERROR

(2189, X'88D') Cluster name resolution failed.

MQRC_CLUSTER_RESOURCE_ERROR

(2269, X'8DD') Cluster resource error.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQR_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.

MQR_DB2_NOT_AVAILABLE
(2342, X'926') Db2 subsystem not available.

MQR_DEF_XMIT_Q_TYPE_ERROR
(2198, X'896') Default transmission queue not local.

MQR_DEF_XMIT_Q_USAGE_ERROR
(2199, X'897') Default transmission queue usage error.

MQR_DYNAMIC_Q_NAME_ERROR
(2011, X'7DB') Name of dynamic queue not valid.

MQR_HANDLE_NOT_AVAILABLE
(2017, X'7E1') No more handles available.

MQR_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQR_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.

MQR_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.

MQR_NAME_IN_USE
(2201, X'899') Name in use.

MQR_NAME_NOT_VALID_FOR_TYPE
(2194, X'892') Object name not valid for object type.

MQR_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.

MQR_OBJECT_ALREADY_EXISTS
(2100, X'834') Object exists.

MQR_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQR_OBJECT_IN_USE
(2042, X'7FA') Object already open with conflicting options.

MQR_OBJECT_LEVEL_INCOMPATIBLE
(2360, X'938') Object level not compatible.

MQR_OBJECT_NAME_ERROR
(2152, X'868') Object name not valid.

MQR_OBJECT_NOT_UNIQUE
(2343, X'927') Object not unique.

MQR_OBJECT_Q_MGR_NAME_ERROR
(2153, X'869') Object queue-manager name not valid.

MQR_OBJECT_RECORDS_ERROR
(2155, X'86B') Object records not valid.

MQR_OBJECT_STRING_ERROR
(2441, X'0989') Objectstring field not valid

MQR_OBJECT_TYPE_ERROR
(2043, X'7FB') Object type not valid.

MQR_OD_ERROR
(2044, X'7FC') Object descriptor structure not valid.

MQR_OPTION_NOT_VALID_FOR_TYPE
(2045, X'7FD') Option not valid for object type.

MQR_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.

MQRC_PAGESET_ERROR

(2193, X'891') Error accessing page-set data set.

MQRC_PAGESET_FULL

(2192, X'890') External storage medium is full.

MQRC_Q_DELETED

(2052, X'804') Queue has been deleted.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_Q_TYPE_ERROR

(2057, X'809') Queue type not valid.

MQRC_RECS_PRESENT_ERROR

(2154, X'86A') Number of records present not valid.

MQRC_REMOTE_Q_NAME_ERROR

(2184, X'888') Remote queue name not valid.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_RESPONSE_RECORDS_ERROR

(2156, X'86C') Response records not valid.

MQRC_SECURITY_ERROR

(2063, X'80F') Security error occurred.

MQRC_SELECTOR_SYNTAX_ERROR

2459 (X'099B') An MQOPEN, MQPUT1 or MQSUB call was issued but a selection string was specified which contained a syntax error.

MQRC_STOPPED_BY_CLUSTER_EXIT

(2188, X'88C') Call rejected by cluster workload exit.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UNKNOWN_ALIAS_BASE_Q

(2082, X'822') Unknown alias base queue.

MQRC_UNKNOWN_DEF_XMIT_Q

(2197, X'895') Unknown default transmission queue.

MQRC_UNKNOWN_OBJECT_NAME

(2085, X'825') Unknown object name.

MQRC_UNKNOWN_OBJECT_Q_MGR

(2086, X'826') Unknown object queue manager.

MQRC_UNKNOWN_REMOTE_Q_MGR

(2087, X'827') Unknown remote queue manager.

MQRC_UNKNOWN_XMIT_Q

(2196, X'894') Unknown transmission queue.

MQRC_WRONG_CF_LEVEL

(2366, X'93E') Coupling-facility structure is wrong level.


MQRC_XMIT_Q_TYPE_ERROR

(2091, X'82B') Transmission queue not local.

MQRC_XMIT_Q_USAGE_ERROR

(2092, X'82C') Transmission queue with wrong usage.

For detailed information about these codes, see:

-  [IBM MQ for z/OS messages, completion, and reason codes](#) for IBM MQ for z/OS.
- [Messages and reason codes](#) for all other IBM MQ platforms except z/OS.

General usage notes

1. The object opened is one of the following:

- A queue to:
 - Get or browse messages (using the MQGET call)
 - Put messages (using the MQPUT call)
 - Inquire about the attributes of the queue (using the MQINQ call)
 - Set the attributes of the queue (using the MQSET call)

If the queue named is a model queue, a dynamic local queue is created. See the *ObjDesc* parameter described in “MQOPEN - Open object” on page 703.

A distribution list is a special type of queue object that contains a list of queues. It can be opened to put messages, but not to get or browse messages, or to inquire or set attributes. See usage note 8 for further details.

A queue that has QSGDISP (GROUP) is a special type of queue definition that cannot be used with the MQOPEN or MQPUT1 calls.

- A namelist to inquire about the names of the queues in the list (using the MQINQ call).
 - A process definition to inquire about the process attributes (using the MQINQ call).
 - The queue manager to inquire about the attributes of the local queue manager (using the MQINQ call).
 - A topic to publish a message (using the MQPUT call)
2. An application can open the same object more than once. A different object handle is returned for each open. Each handle that is returned can be used for the functions for which the corresponding open was performed.
3. If the object being opened is a queue other than a cluster queue, all name resolution within the local queue manager takes place at the time of the MQOPEN call. This can include:
- Resolution of the name of a local definition of a remote queue to the name of the remote queue manager, and the name by which the queue is known at the remote queue manager
 - Resolution of the remote queue-manager name to the name of a local transmission queue
 - (z/OS only) Resolution of the remote queue-manager name to the name of the shared transmission queue used by the IGQ agent (applies only if the local and remote queue managers belong to the same queue-sharing group)
 - Alias resolution to the name of a base queue or a topic object.

However, be aware that subsequent MQINQ or MQSET calls for the handle relate solely to the name that has been opened, and not to the object resulting after name resolution has occurred. For example,

if the object opened is an alias, the attributes returned by the MQINQ call are the attributes of the alias, not the attributes of the base queue or a topic object to which the alias resolves.

If the object being opened is a cluster queue, name resolution can occur at the time of the MQOPEN call, or be deferred until later. The point at which resolution occurs is controlled by the MQOO_BIND_* options specified on the MQOPEN call:

- MQOO_BIND_ON_OPEN
- MQOO_BIND_NOT_FIXED
- MQOO_BIND_AS_Q_DEF
- MQOO_BIND_ON_GROUP


See [Name resolution](#) for more information about name resolution for cluster queues.

4. An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, for use with MQGET calls that specify the object handle and one of the browse options. This allows the queue to be scanned without altering its contents. A message that has been found by browsing can be removed from the queue by using the MQGMO_MSG_UNDER_CURSOR option.

Multiple browse cursors can be active for a single application by issuing several MQOPEN requests for the same queue.

5. Applications started by a trigger monitor are passed the name of the queue that is associated with the application when the application is started. This queue name can be specified in the *ObjDesc* parameter to open the queue. See [“MQTMC2 - Trigger message 2 \(character format\)”](#) on page 579 for further details.
6. On IBM i, applications running in compatibility mode are connected automatically to the queue manager by the first MQOPEN call issued by the application (if the application has not already connected to the queue manager by using the MQCONN call).

Applications not running in compatibility mode must issue the MQCONN or MQCONNX call to connect to the queue manager explicitly, before using the MQOPEN call to open an object.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Read ahead options

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- Both the client and remote queue manager must be at WebSphere MQ Version 7 or later.
- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.
- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

The following notes apply to the use of read ahead options.

1. The read ahead options are applicable only when one, and only one, of the MQOO_BROWSE, MQOO_INPUT_SHARED and MQOO_INPUT_EXCLUSIVE options are also specified. An error is not thrown if a read ahead options are specified with the MQOO_INQUIRE or MQOO_SET options.
2. Read ahead is not enabled when requested if the options used on the first MQGET call are not supported for use with read ahead. Also, read ahead is disabled when the client is connecting to a queue manager that does not support read ahead.
3. If the application is not running as an IBM MQ client, read ahead options are ignored.

Cluster queues

The following notes apply to the use of cluster queues.

1. When a cluster queue is opened for the first time, and the local queue manager is not a full repository queue manager, the local queue manager obtains information about the cluster queue from a full repository queue manager. When the network is busy, it can take several seconds for the local queue manager to receive the needed information from the repository queue manager. As a result, the application issuing the MQOPEN call might have to wait for up to 10 seconds before control returns from the MQOPEN call. If the local queue manager does not receive the needed information about the cluster queue within this time, the call fails with reason code MQRC_CLUSTER_RESOLUTION_ERROR.
2. When a cluster queue is opened and there are multiple instances of the queue in the cluster, the instance opened depends on the options specified on the MQOPEN call:
 - If the options specified include any of the following:
 - MQOO_BROWSE
 - MQOO_INPUT_AS_Q_DEF
 - MQOO_INPUT_EXCLUSIVE
 - MQOO_INPUT_SHARED
 - MQOO_SET

the instance of the cluster queue opened must be the local instance. If there is no local instance of the queue, the MQOPEN call fails.
 - If the options specified include none of the options described previously, but include one or both of the following:
 - MQOO_INQUIRE
 - MQOO_OUTPUT

the instance opened is the local instance if there is one, and a remote instance otherwise (if using the CLWLUSEQ defaults). The instance chosen by the queue manager can, however, be altered by a cluster workload exit (if there is one).
3. If there is a subscription for the queue, but it is not acknowledged by a full repository, the object is not present in the cluster and the call fails with reason code MQRC_OBJECT_NAME.

For more information about cluster queues, see [Cluster queues](#).

Distribution lists

The following notes apply to the use of distribution lists.

Distribution lists are supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

1. Fields in the MQOD structure must be set as follows when opening a distribution list:
 - *Version* must be MQOD_VERSION_2 or greater.
 - *ObjectType* must be MQOT_Q.
 - *ObjectName* must be blank or the null string.
 - *ObjectQMgrName* must be blank or the null string.
 - *RecsPresent* must be greater than zero.
 - One of *ObjectRecOffset* and *ObjectRecPtr* must be zero and the other nonzero.
 - No more than one of *ResponseRecOffset* and *ResponseRecPtr* can be nonzero.
 - There must be *RecsPresent* object records, addressed by either *ObjectRecOffset* or *ObjectRecPtr*. The object records must be set to the names of the destination queues to be opened.
 - If one of *ResponseRecOffset* and *ResponseRecPtr* is nonzero, there must be *RecsPresent* response records present. They are set by the queue manager if the call completes with reason code MQRC_MULTIPLE_REASONS.

A version-2 MQOD can also be used to open a single queue that is not in a distribution list, by ensuring that *RecsPresent* is zero.

2. Only the following open options are valid in the *Options* parameter:

- MQOO_OUTPUT
- MQOO_PASS_*_CONTEXT
- MQOO_SET_*_CONTEXT
- MQOO_ALTERNATE_USER_AUTHORITY
- MQOO_FAIL_IF QUIESCING

3. The destination queues in the distribution list can be local, alias, or remote queues, but they cannot be model queues. If a model queue is specified, that queue fails to open, with reason code MQRC_Q_TYPE_ERROR. However, this does not prevent other queues in the list being opened successfully.

4. The completion code and reason code parameters are set as follows:

- If the open operations for the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

For example, if every open succeeds, the completion code is set to MQCC_OK and the reason code is set to MQRC_NONE; if every open fails because none of the queues exists, the parameters are set to MQCC_FAILED and MQRC_UNKNOWN_OBJECT_NAME.

- If the open operations for the queues in the distribution list do not all succeed or fail in the same way:
 - The completion code parameter is set to MQCC_WARNING if at least one open succeeded, and to MQCC_FAILED if all failed.
 - The reason code parameter is set to MQRC_MULTIPLE_REASONS.
 - The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.

5. When a distribution list has been opened successfully, the handle *Hobj* returned by the call can be used on subsequent MQPUT calls to put messages to queues in the distribution list, and on an MQCLOSE call to relinquish access to the distribution list. The only valid close option for a distribution list is MQCO_NONE.

The MQPUT1 call can also be used to put a message to a distribution list; the MQOD structure defining the queues in the list is specified as a parameter on that call.

6. Each successfully opened destination in the distribution list counts as a separate handle when checking whether the application has exceeded the permitted maximum number of handles (see the *MaxHandles* queue-manager attribute). This is true even when two or more of the destinations in the distribution list resolve to the same physical queue. If the MQOPEN or MQPUT1 call for a distribution list would cause the number of handles in use by the application to exceed *MaxHandles*, the call fails with reason code MQRC_HANDLE_NOT_AVAILABLE.

7. Each destination that is opened successfully has the value of its *OpenOutputCount* attribute incremented by one. If two or more of the destinations in the distribution list resolve to the same physical queue, that queue has its *OpenOutputCount* attribute incremented by the number of destinations in the distribution list that resolve to that queue.

8. Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.

9. A distribution list can contain only one destination.

Remote queues

The following notes apply to the use of remote queues.

A remote queue can be specified in one of two ways in the *ObjDesc* parameter of this call.

- By specifying for *ObjectName* the name of a local definition of the remote queue. In this case, *ObjectQMGrName* refers to the local queue manager, and can be specified as blanks or (in the C programming language) a null string.

The security validation performed by the local queue manager verifies that the user is authorized to open the local definition of the remote queue.

- By specifying for *ObjectName* the name of the remote queue as known to the remote queue manager. In this case, *ObjectQMGrName* is the name of the remote queue manager.

The security validation performed by the local queue manager verifies that the user is authorized to send messages to the transmission queue resulting from the name resolution process.

In either case:

- No messages are sent by the local queue manager to the remote queue manager to check that the user is authorized to put messages on the queue.
- When a message arrives at the remote queue manager, the remote queue manager might reject it because the user originating the message is not authorized.

See the *ObjectName* and *ObjectQMGrName* fields described in [“MQOD - Object descriptor” on page 455](#) for more information.

Objects

Security

The following notes relate to the security aspects of using MQOPEN.

The queue manager performs security checks when an MQOPEN call is issued, to verify that the user identifier under which the application is running has the appropriate level of authority before access is permitted. The authority check is made on the name of the object being opened, and not on the name, or names, resulting after a name has been resolved.

If the object being opened is an alias queue which points at a topic object, the queue manager performs a security check on the alias queue name, before performing a security check for the topic as if the topic object had been used directly.

If the object being opened is a topic object, whether with *ObjectName* alone or by using the *ObjectString* (with or without a basing *ObjectName*), the queue manager performs the security check by using the resultant topic string, taken from within the topic object specified in *ObjectName*, and if required concatenating it with that provided in *ObjectString*, and then finding the closest topic object at or above that point in the topic tree to perform the security check against. This might not be the same topic object that was specified in *ObjectName*.

If the object being opened is a model queue, the queue manager performs a full security check against both the name of the model queue and the name of the dynamic queue that is created. If the resulting dynamic queue is then opened explicitly, a further resource security check is performed against the name of the dynamic queue.

On z/OS, the queue manager performs security checks only if security is enabled. For more information about security checking, see [Setting up security on z/OS](#).

Attributes

The following notes relate to attributes.

The attributes of an object can change while an application has the object open. In many cases, the application does not notice this, but for certain attributes the queue manager marks the handle as no longer valid. These attributes are:

- Any attribute that affects the name resolution of the object. This applies regardless of the open options used, and includes the following:
 - A change to the *BaseQName* attribute of an alias queue that is open.
 - A change to the *TargetType* attribute of an alias queue that is open.
 - A change to the *RemoteQName* or *RemoteQMGrName* queue attributes, for any handle that is open for this queue, or for a queue that resolves through this definition as a queue-manager alias.
 - Any change that causes a currently open handle for a remote queue to resolve to a different transmission queue, or to fail to resolve to one at all. For example, this can include:
 - A change to the *XmitQName* attribute of the local definition of a remote queue, whether the definition is being used for a queue, or for a queue-manager alias.
 - (z/OS only) A change to the value of the *IntraGroupqueuing* queue-manager attribute, or a change in the definition of the shared transmission queue (SYSTEM.QSG.TRANSMIT.QUEUE) used by the IGQ agent.

There is one exception to this: the creation of a new transmission queue. A handle that would have resolved to this queue had it been present when the handle was opened, but instead resolved to the default transmission queue, is not made invalid.

- A change to the *DefXmitQName* queue-manager attribute. In this case all open handles that resolved to the previously named queue (that resolved to it only because it was the default transmission queue) are marked as invalid. Handles that resolved to this queue for other reasons are not affected.
- The *Shareability* queue attribute, if there are two or more handles that are currently providing MQOO_INPUT_SHARED access for this queue, or for a queue that resolves to this queue. If so, *all* handles that are open for this queue, or for a queue that resolves to this queue, are marked as invalid, regardless of the open options.

On z/OS, the handles previously described are marked as invalid if one or more handles is currently providing MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE access to the queue.

- The *Usage* queue attribute, for all handles that are open for this queue, or for a queue that resolves to this queue, regardless of the open options.

When a handle is marked as invalid, all subsequent calls (other than MQCLOSE) using this handle fail with reason code MQRC_OBJECT_CHANGED. The application must issue an MQCLOSE call (using the original handle) and then reopen the queue. Any uncommitted updates against the old handle from previous successful calls can still be committed or backed out, as required by the application logic.

If changing an attribute causes this to happen, use a special force version of the call.

C invocation

```
MQOPEN (Hconn, &ObjDesc, Options, &Hobj, &CompCode,
        &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;      /* Connection handle */
MQOD      ObjDesc;    /* Object descriptor */
MQLONG    Options;    /* Options that control the action of MQOPEN */
MQHOBJ     Hobj;      /* Object handle */
MQLONG    CompCode;   /* Completion code */
MQLONG    Reason;     /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQOPEN' USING HCONN, OBJDESC, OPTIONS, HOBJ, COMPCODE, REASON
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN      PIC S9(9) BINARY.
** Object descriptor
01 OBJDESC.
   COPY CMQODV.
** Options that control the action of MQOPEN
01 OPTIONS    PIC S9(9) BINARY.
** Object handle
01 HOBJ       PIC S9(9) BINARY.
** Completion code
01 COMPCODE   PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON     PIC S9(9) BINARY.
```

PL/I invocation

```
call MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */
dcl ObjDesc    like MQOD;    /* Object descriptor */
dcl Options    fixed bin(31); /* Options that control the action of
                               MQOPEN */
dcl Hobj       fixed bin(31); /* Object handle */
dcl CompCode   fixed bin(31); /* Completion code */
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQOPEN,(HCONN,OBJDESC,OPTIONS,HOBJ,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN      DS      F  Connection handle
OBJDESC    CMQODA   ,  Object descriptor
OPTIONS    DS      F  Options that control the action of MQOPEN
HOBJ       DS      F  Object handle
COMPCODE   DS      F  Completion code
REASON     DS      F  Reason code qualifying COMPCODE
```

Visual Basic invocation

```
MQOPEN Hconn, ObjDesc, Options, Hobj, CompCode, Reason
```

Declare the parameters as follows:

```
Dim Hconn As Long 'Connection handle'
Dim ObjDesc As MQOD 'Object descriptor'
Dim Options As Long 'Options that control the action of MQOPEN'
Dim Hobj As Long 'Object handle'
Dim CompCode As Long 'Completion code'
Dim Reason As Long 'Reason code qualifying CompCode'
```

MQPUT - Put message

The MQPUT call puts a message on a queue or distribution list, or to a topic. The queue, distribution list, or topic must already be open.

Syntax

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason)
```

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Hobj

Type: MQHOBJ - input

This handle represents the queue to which the message is added, or the topic to which the message is published. The value of *Hobj* was returned by a previous MQOPEN call that specified the MQOO_OUTPUT option.

MsgDesc

Type: MQMD - input/output

This structure describes the attributes of the message being sent, and receives information about the message after the put request is complete. See [“MQMD - Message descriptor” on page 396](#) for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure to specify values for the fields that exist in the version-2 MQMD but not the version-1. The *Format* field in the MQMD must be set to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. See [“MQMDE - Message descriptor extension” on page 447](#) for more details.

The application does not need to provide an MQMD structure if a valid message handle is supplied in the *OriginalMsgHandle* or *NewMsgHandle* fields of the MQPMO structure. If nothing is provided in one of these fields, the descriptor of the message is taken from the descriptor associated with the message handles.

If you use, or plan to use, API exits then we recommend that you explicitly supply an MQMD structure and do not use the message descriptors associated with the message handles. This is because the API Exit associated with MQPUT or MQPUT1 call is unable to ascertain which MQMD values are used by the queue manager to complete the MQPUT or MQPUT1 request.

PutMsgOpts

Type: MQPMO - input/output

See [“MQPMO - Put-message options” on page 475](#) for details.

BufferLength

Type: MQLONG - input

The length of the message in *Buffer*. Zero is valid, and indicates that the message contains no application data. The upper limit for *BufferLength* depends on various factors:

- If the destination is a local queue or resolves to a local queue, the upper limit depends on whether:
 - The local queue manager supports segmentation.
 - The sending application specifies the flag that allows the queue manager to segment the message. This flag is MQMF_SEGMENTATION_ALLOWED, and can be specified either in a version-2 MQMD, or in an MQMDE used with a version-1 MQMD.

If both of these conditions are satisfied, *BufferLength* cannot exceed 999 999 999 minus the value of the *Offset* field in MQMD. The longest logical message that can be put is therefore 999 999 999 bytes (when *Offset* is zero). However, resource constraints imposed by the operating system or environment in which the application is running might result in a lower limit.

If one or both of the previous conditions is not satisfied, *BufferLength* cannot exceed the smaller of the queue's *MaxMsgLength* attribute and queue-manager's *MaxMsgLength* attribute.

- If the destination is a remote queue or resolves to a remote queue, the conditions for local queues apply, *but at each queue manager through which the message must pass in order to reach the destination queue*; in particular:
 1. The local transmission queue used to store the message temporarily at the local queue manager
 2. Intermediate transmission queues (if any) used to store the message at queue managers on the route between the local and destination queue managers
 3. The destination queue at the destination queue manager

The longest message that can be put is therefore governed by the most restrictive of these queues and queue managers.

When a message is on a transmission queue, additional information resides with the message data, and this reduces the amount of application data that can be carried. In this situation, subtract MQ_MSG_HEADER_LENGTH bytes from the *MaxMsgLength* values of the transmission queues when determining the limit for *BufferLength*.

Note: Only failure to comply with condition 1 can be diagnosed synchronously (with reason code MQRC_MSG_TOO_BIG_FOR_Q or MQRC_MSG_TOO_BIG_FOR_Q_MGR) when the message is put. If conditions 2 or 3 are not satisfied, the message is redirected to a dead-letter (undelivered-message) queue, either at an intermediate queue manager or at the destination queue manager. If this happens, a report message is generated if one was requested by the sender.

Buffer

Type: MQBYTEExBufferLength - input

This is a buffer containing the application data to be sent. The buffer must be aligned on a boundary appropriate to the nature of the data in the message. 4-byte alignment is suitable for most messages (including messages containing IBM MQ header structures), but some messages might require more stringent alignment. For example, a message containing a 64-bit binary integer might require 8-byte alignment.

If *Buffer* contains character or numeric data, set the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to the values appropriate to the data; this enables the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All the other parameters on the MQPUT call must be in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE).

In the C programming language, the parameter is declared as a pointer-to-void; the address of any type of data can be specified as the parameter.

If the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_INCOMPLETE_GROUP

(2241, X'8C1') Message group not complete.

MQRC_INCOMPLETE_MSG

(2242, X'8C2') Logical message not complete.

MQRC_INCONSISTENT_PERSISTENCE

(2185, X'889') Inconsistent persistence specification.

MQRC_INCONSISTENT_UOW

(2245, X'8C5') Inconsistent unit-of-work specification.

MQRC_MULTIPLE_REASONS

(2136, X'858') Multiple reason codes returned.

MQRC_PRIORITY_EXCEEDS_MAXIMUM

(2049, X'801') Message Priority exceeds maximum value supported.

MQRC_UNKNOWN_REPORT_OPTION

(2104, X'838') Report option(s) in message descriptor not recognized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ALIAS_TARGTYPE_CHANGED

(2480, X'09B0') Subscription target type has changed from queue to topic object.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BACKED_OUT

(2003, X'7D3') Unit of work backed out.

MQRC_BUFFER_ERROR

(2004, X'7D4') Buffer parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CALL_INTERRUPTED

(2549, X'9F5') MQPUT or MQCMIT was interrupted and reconnection processing cannot reestablish a definite outcome.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') Coupling facility not available.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CFGR_ERROR

(2416, X'970') PCF group parameter structure MQCFGR in the message data is not valid.

MQRC_CFH_ERROR

(2235, X'8BB') PCF header structure not valid.

MQRC_CFIF_ERROR

(2414, X'96E') PCF integer filter parameter structure in the message data is not valid.

MQRC_CFIL_ERROR

(2236, X'8BC') PCF integer list parameter structure or PCIF*64 integer list parameter structure not valid.

MQRC_CFIN_ERROR

(2237, X'8BD') PCF integer parameter structure or PCIF*64 integer parameter structure not valid.

MQRC_CFSF_ERROR

(2415, X'96F') PCF string filter parameter structure in the message data is not valid.

MQRC_CFSL_ERROR

(2238, X'8BE') PCF string list parameter structure not valid.

MQRC_CFST_ERROR

(2239, X'8BF') PCF string parameter structure not valid.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CLUSTER_EXIT_ERROR

(2266, X'8DA') Cluster workload exit failed.

MQRC_CLUSTER_RESOLUTION_ERROR

(2189, X'88D') Cluster name resolution failed.

MQRC_CLUSTER_RESOURCE_ERROR

(2269, X'8DD') Cluster resource error.

MQRC_COD_NOT_VALID_FOR_XCF_Q

(2106, X'83A') COD report option not valid for XCF queue.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CONTENT_ERROR

2554 (X'09FA') Message content could not be parsed to determine whether the message should be delivered to a subscriber with an extended message selector.

MQRC_CONTEXT_HANDLE_ERROR

(2097, X'831') Queue handle referred to does not save context.

MQRC_CONTEXT_NOT_AVAILABLE

(2098, X'832') Context not available for queue handle referred to.

MQRC_DATA_LENGTH_ERROR

(2010, X'7DA') Data length parameter not valid.

MQRC_DH_ERROR

(2135, X'857') Distribution header structure not valid.

MQRC_DLH_ERROR
(2141, X'85D') Dead letter header structure not valid.

MQRC_EPH_ERROR
(2420, X'974') Embedded PCF structure not valid.

MQRC_EXPIRY_ERROR
(2013, X'7DD') Expiry time not valid.

MQRC_FEEDBACK_ERROR
(2014, X'7DE') Feedback code not valid.

MQRC_GLOBAL_UOW_CONFLICT
(2351, X'92F') Global units of work conflict.

MQRC_GROUP_ID_ERROR
(2258, X'8D2') Group identifier not valid.

MQRC_HANDLE_IN_USE_FOR_UOW
(2353, X'931') Handle in use for global unit of work.

MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQRC_HEADER_ERROR
(2142, X'85E') MQ header structure not valid.

MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.

MQRC_IIH_ERROR
(2148, X'864') IMS information header structure not valid.

MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.

MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.

MQRC_INCONSISTENT_PERSISTENCE
(2185, X'889') Inconsistent persistence specification.

MQRC_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.

MQRC_LOCAL_UOW_CONFLICT
(2352, X'930') Global unit of work conflicts with local unit of work.

MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.

MQRC_MDE_ERROR
(2248, X'8C8') Message descriptor extension not valid.

MQRC_MISSING_REPLY_TO_Q
(2027, X'7EB') Missing reply-to queue or MQPMO_SUPPRESS_REPLYTO was used

MQRC_MISSING_WIH
(2332, X'91C') Message data does not begin with MQWIH.

MQRC_MSG_FLAGS_ERROR
(2249, X'8C9') Message flags not valid.

MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.

MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.

MQRC_MSG_TOO_BIG_FOR_Q_MGR
(2031, X'7EF') Message length greater than maximum for queue manager.

MQRC_MSG_TYPE_ERROR
(2029, X'7ED') Message type in message descriptor not valid.

MQRC_MULTIPLE_REASONS

(2136, X'858') Multiple reason codes returned.

MQRC_NO_DESTINATIONS_AVAILABLE

(2270, X'8DE') No destination queues available.

MQRC_NOT_OPEN_FOR_OUTPUT

(2039, X'7F7') Queue not open for output.

MQRC_NOT_OPEN_FOR_PASS_ALL

(2093, X'82D') Queue not open for pass all context.

MQRC_NOT_OPEN_FOR_PASS_IDENT

(2094, X'82E') Queue not open for pass identity context.

MQRC_NOT_OPEN_FOR_SET_ALL

(2095, X'82F') Queue not open for set all context.

MQRC_NOT_OPEN_FOR_SET_IDENT

(2096, X'830') Queue not open for set identity context.

MQRC_OBJECT_CHANGED

(2041, X'7F9') Object definition changed since opened.

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged.

MQRC_OFFSET_ERROR

(2251, X'8CB') Message segment offset not valid.

MQRC_OPEN_FAILED

(2137, X'859') Object not opened successfully.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_ORIGINAL_LENGTH_ERROR

(2252, X'8CC') Original length not valid.

MQRC_PAGESET_ERROR

(2193, X'891') Error accessing page-set data set.

MQRC_PAGESET_FULL

(2192, X'890') External storage medium is full.

MQRC_PCF_ERROR

(2149, X'865') PCF structures not valid.

MQRC_PERSISTENCE_ERROR

(2047, X'7FF') Persistence not valid.

MQRC_PERSISTENT_NOT_ALLOWED

(2048, X'800') Queue does not support persistent messages.

MQRC_PMO_ERROR

(2173, X'87D') Put-message options structure not valid.

MQRC_PMO_RECORD_FLAGS_ERROR

(2158, X'86E') Put message record flags not valid.

MQRC_PRIORITY_ERROR

(2050, X'802') Message priority not valid.

MQRC_PUBLICATION_FAILURE

(2502, X'9C6') The publication has not been delivered to any of the subscribers.

MQRC_PUT_INHIBITED

(2051, X'803') Put calls inhibited for the queue, for the queue to which this queue resolves, or the topic.

MQRC_PUT_MSG_RECORDS_ERROR

(2159, X'86F') Put message records not valid.

MQRC_PUT_NOT_RETAINED

(2479, X'09AF') Publication could not be retained

MQRC_Q_DELETED

(2052, X'804') Queue has been deleted.

MQRC_Q_FULL

(2053, X'805') Queue already contains maximum number of messages.

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down.

MQRC_Q_SPACE_NOT_AVAILABLE

(2056, X'808') No space available on disk for queue.

MQRC_RECONNECT_FAILED

(2548, X'9F4') After reconnecting, an error occurred reinstating the handles for a reconnectable connection.

MQRC_RECS_PRESENT_ERROR

(2154, X'86A') Number of records present not valid.

MQRC_REPORT_OPTIONS_ERROR

(2061, X'80D') Report options in message descriptor not valid.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_RESPONSE_RECORDS_ERROR

(2156, X'86C') Response records not valid.

MQRC_RFH_ERROR

(2334, X'91E') MQRFH or MQRFH2 structure not valid.

MQRC_RMH_ERROR

(2220, X'8AC') Reference message header structure not valid.

MQRC_SEGMENT_LENGTH_ZERO

(2253, X'8CD') Length of data in message segment is zero.

MQRC_SEGMENTS_NOT_SUPPORTED

(2365, X'93D') Segments not supported.

MQRC_SELECTION_NOT_AVAILABLE

2551 (X'09F7') A possible subscriber for the publication exists, but the queue manager cannot check whether to send the publication to the subscriber.

MQRC_STOPPED_BY_CLUSTER_EXIT

(2188, X'88C') Call rejected by cluster workload exit.

MQRC_STORAGE_CLASS_ERROR

(2105, X'839') Storage class error.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_SYNCPOINT_LIMIT_REACHED

(2024, X'7E8') No more messages can be handled within current unit of work.

MQRC_SYNCPOINT_NOT_AVAILABLE

(2072, X'818') Syncpoint support not available.

MQRC_TM_ERROR

(2265, X'8D9') Trigger message structure not valid.

MQRC_TMC_ERROR

(2191, X'88F') Character trigger message structure not valid.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UOW_ENLISTMENT_ERROR

(2354, X'932') Enlistment in global unit of work failed.

MQRC_UOW_MIX_NOT_SUPPORTED

(2355, X'933') Mixture of unit-of-work calls not supported.

MQRC_UOW_NOT_AVAILABLE

(2255, X'8CF') Unit of work not available for the queue manager to use.

MQRC_WIH_ERROR

(2333, X'91D') MQWIH structure not valid.

MQRC_WRONG_MD_VERSION

(2257, X'8D1') Wrong version of MQMD supplied.

MQRC_XQH_ERROR

(2260, X'8D4') Transmission queue header structure not valid.

For detailed information about these codes, see [Messages and reason codes](#).

Topic usage notes

1. The following notes apply to the use of topics:

- a. When using MQPUT to publish messages on a topic, where one or more subscribers to that topic cannot be given the publication due to a problem with their subscriber queue (for example it is full), the Reason code returned to the MQPUT call and the delivery behavior is dependent on the setting of the PMSGDLV or NPMSGDLV attributes on the TOPIC. Note delivery of a publication to the dead letter queue when MQRO_DEAD_LETTER_Q is specified, or discarding the message when MQRO_DISCARD_MSG is specified, is considered as a successful delivery of the message. If none of the publications are delivered, the MQPUT returns with MQRC_PUBLICATION_FAILURE. This can happen in the following cases:

- A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALL and any subscription (durable or not) has a queue which cannot receive the publication.
- A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLDUR and a durable subscription has a queue which cannot receive the publication.

The MQPUT can return with MQRC_NONE even though publications could not be delivered to some subscribers in the following cases:

- A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLAVAIL and any subscription, durable or not, has a queue which cannot receive the publication.
- A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLDUR and a non-durable subscription has a queue which cannot receive the publication.

You can use the USEDQLQ topic attribute to determine whether the dead-letter queue is used when publication messages cannot be delivered to their correct subscriber queue. For more information about the use of USEDQLQ, see [DEFINE TOPIC](#).

- b. If there are no subscribers to the topic being used, the message published is not sent to any queue and is discarded. It does not matter whether the message is persistent or non-persistent, or whether it has unlimited expiry or has an expiry time, it is still discarded if there are no subscribers. The exception to this is if the message is to be retained, in which case, although it is not sent to any subscribers' queues, it is stored against the topic to be delivered to any new subscriptions or to any subscribers that ask for retained publications using MQSUBRQ.

MQPUT and MQPUT1

You can use both the MQPUT and MQPUT1 calls to put messages on a queue; which call to use depends on the circumstances

- Use the MQPUT call to place multiple messages on the *same* queue.

An MQOPEN call specifying the MQOO_OUTPUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.

- Use the MQPUT1 call to put only *one* message on a queue.

This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, minimizing the number of calls that must be issued.

Destination Queues

The following notes apply to the use of destination queues:

1. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if the conditions detailed are satisfied. Some conditions apply to both local and remote destination queues; other conditions apply only to remote destination queues.

Conditions that apply to local and remote destination queues

- All the MQPUT calls are within the same unit of work, or none of them is within a unit of work.

Be aware that when messages are put onto a particular queue within a single unit of work, messages from other applications might be interspersed with the sequence of messages on the queue.

- All the MQPUT calls are made using the same object handle *Hobj*.

In some environments, message sequence is also preserved when different object handles are used, if the calls are made from the same application. The meaning of *same application* is determined by the environment:

- On z/OS, the application is:
 - For CICS, the CICS task
 - For IMS, the task
 - For z/OS batch, the task
- On IBM i, the application is the job.
- On Windows and UNIX systems, the application is the thread.

- The messages all have the same priority.
- The messages are not put to a cluster queue with MQOO_BIND_NOT_FIXED specified (or with MQOO_BIND_AS_Q_DEF in effect when the DefBind queue attribute has the value MQBND_BIND_NOT_FIXED).

Additional conditions that apply to remote destination queues

- There is only one path from the sending queue manager to the destination queue manager.

If some messages in the sequence might go on a different path (for example, because of reconfiguration, traffic balancing, or path selection based on message size), the order of the messages at the destination queue manager cannot be guaranteed.

- Messages are not placed temporarily on dead-letter queues at the sending, intermediate, or destination queue managers.

If one or more of the messages is put temporarily on a dead-letter queue (for example, because a transmission queue or the destination queue is temporarily full), the messages can arrive on the destination queue out of sequence.

- The messages are either all persistent or all nonpersistent.

If a channel on the route between the sending and destination queue managers has its *NonPersistentMsgSpeed* attribute set to *MQNPMS_FAST*, nonpersistent messages can jump ahead of persistent messages, resulting in the order of persistent messages relative to nonpersistent messages not being preserved. However, the order of persistent messages relative to each other, and of nonpersistent messages relative to each other, is preserved.

If these conditions are not satisfied, you can use message groups to preserve message order, but this requires both the sending and receiving applications to use the message-grouping support. For more information about message groups, see:

- [MQMD - MsgFlags field](#)
- [MQPMO_LOGICAL_ORDER](#)
- [MQGMO_LOGICAL_ORDER](#)

Distribution Lists

The following notes apply to the use of distribution lists.

Distribution lists are supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

1. You can put messages to a distribution list using either a version-1 or a version-2 MQPMO. If you use a version-1 MQPMO (or a version-2 MQPMO with *RecsPresent* equal to zero), the application can provide no put message records or response records. You cannot identify the queues that encounter errors if the message is sent successfully to some queues in the distribution list and not others.

If the application provides put message records or response records, set the *Version* field to *MQPMO_VERSION_2*.

You can also use a version-2 MQPMO to send messages to a single queue that is not in a distribution list, by ensuring that *RecsPresent* is zero.

2. The completion code and reason code parameters are set as follows:

- If the puts to the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

For example, if every put succeeds, the completion code and reason code are set to *MQCC_OK* and *MQRC_NONE*; if every put fails because all the queues are inhibited for puts, the parameters are set to *MQCC_FAILED* and *MQRC_PUT_INHIBITED*.

- If the puts to the queues in the distribution list do not all succeed or fail in the same way:
 - The completion code parameter is set to *MQCC_WARNING* if at least one put succeeded, and to *MQCC_FAILED* if all failed.
 - The reason code parameter is set to *MQRC_MULTIPLE_REASONS*.
 - The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.

If the put to a destination fails because the open for that destination failed, the fields in the response record are set to *MQCC_FAILED* and *MQRC_OPEN_FAILED*; that destination is included in *InvalidDestCount*.

3. If a destination in the distribution list resolves to a local queue, the message is placed on that queue in normal form (that is, not as a distribution-list message). If more than one destination resolves to the same local queue, one message is placed on the queue for each such destination.

If a destination in the distribution list resolves to a remote queue, a message is placed on the appropriate transmission queue. Where several destinations resolve to the same transmission queue, a single distribution-list message containing those destinations can be placed on the transmission queue, even if those destinations were not adjacent in the list of destinations provided by the application. However, this can be done only if the transmission queue supports distribution-list messages (see [DistLists](#)).

If the transmission queue does not support distribution lists, one copy of the message in normal form is placed on the transmission queue for each destination that uses that transmission queue.

If a distribution list with the application message data is too large for a transmission queue, the distribution list message is split into smaller distribution-list messages, each containing fewer destinations. If the application message data only just fits on the queue, distribution-list messages cannot be used at all, and the queue manager generates one copy of the message in normal form for each destination that uses that transmission queue.

If different destinations have different message priority or message persistence (this can occur when the application specifies MQPRI_PRIORITY_AS_Q_DEF or MQPER_PERSISTENCE_AS_Q_DEF), the messages are not held in the same distribution-list message. Instead, the queue manager generates as many distribution-list messages as are necessary to accommodate the differing priority and persistence values.

4. A put to a distribution list can result in:

- A single distribution-list message, or
- A number of smaller distribution-list messages, or
- A mixture of distribution list messages and normal messages, or
- Normal messages only.

Which of the above occurs depends on whether:

- The destinations in the list are local, remote, or a mixture.
- The destinations have the same message priority and message persistence.
- The transmission queues can hold distribution-list messages.
- The transmission queues' maximum message lengths are large enough to accommodate the message in distribution-list form.

However, regardless of which of the above occurs, each *physical* message resulting (that is, each normal message or distribution-list message resulting from the put) counts as only *one* message when:

- Checking whether the application has exceeded the permitted maximum number of messages in a unit of work (see the *MaxUncommittedMsgs* queue-manager attribute).
- Checking whether the triggering conditions are satisfied.
- Incrementing queue depths and checking whether the queues' maximum queue depth would be exceeded.

5. Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.

Headers

If a message is put with one or more IBM MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. If the queue manager detects an error, the call fails with an appropriate reason code. The checks performed vary according to the particular structures that are present:

- Checks are performed only if a version-2 or later MQMD is used on the MQPUT or MQPUT1 call. Checks are not performed if a version-1 MQMD is used, even if an MQMDE is present at the start of the message data.
- Structures that are not supported by the local queue manager, and structures following the first MQDLH in the message, are not validated.
- The MQDH and MQMDE structures are validated completely by the queue manager.
- Other structures are validated partially by the queue manager (not all fields are checked).

General checks performed by the queue manager include the following:

- The *StrucId* field must be valid.
- The *Version* field must be valid.
- The *StrucLength* field must specify a value that is large enough to include the structure plus any variable-length data that forms part of the structure.
- The *CodedCharSetId* field must not be zero, or a negative value that is not valid (MQCCSI_DEFAULT, MQCCSI_EMBEDDED, MQCCSI_Q_MGR, and MQCCSI_UNDEFINED are *not* valid in most IBM MQ header structures).
- The *BufferLength* parameter of the call must specify a value that is large enough to include the structure (the structure must not extend beyond the end of the message).

In addition to general checks on structures, the following conditions must be satisfied:

- The sum of the lengths of the structures in a PCF message must equal the length specified by the *BufferLength* parameter on the MQPUT or MQPUT1 call. A PCF message is a message that has a format name of MQFMT_ADMIN, MQFMT_EVENT, or MQFMT_PCF.
- An IBM MQ structure must not be truncated, except in the following situations where truncated structures are permitted:
 - Messages that are report messages.
 - PCF messages.
 - Messages containing an MQDLH structure. (Structures *following* the first MQDLH can be truncated; structures preceding the MQDLH cannot.)
- An IBM MQ structure must not be split over two or more segments; the structure must be contained entirely within one segment.

Buffer

For the Visual Basic programming language, the following points apply:

- If the size of the *Buffer* parameter is less than the length specified by the *BufferLength* parameter, the call fails with reason code MQRC_BUFFER_LENGTH_ERROR.
- The *Buffer* parameter is declared as being of type *String*. If the data to be placed on the queue is not of type *String*, use the MQPUTAny call in place of MQPUT.

The MQPUTAny call has the same parameters as the MQPUT call, except that the *Buffer* parameter is declared as being of type *Any*, allowing any type of data to be placed on the queue. However, this means that *Buffer* cannot be checked to ensure that it is at least *BufferLength* bytes in size.

C invocation

```
MQPUT (Hconn, Hobj, &MsgDesc, &PutMsgOpts, BufferLength, Buffer,
       &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQHOBJ   Hobj;            /* Object handle */
MQMD     MsgDesc;         /* Message descriptor */
```

```

MQPMO    PutMsgOpts;      /* Options that control the action of MQPUT */
MQLONG   BufferLength;    /* Length of the message in Buffer */
MQBYTE   Buffer[n];       /* Message data */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying CompCode */

```

COBOL invocation

```

CALL 'MQPUT' USING HCONN, HOBJ, MSGDESC, PUTMSGOPTS, BUFFERLENGTH,
                  BUFFER, COMPCODE, REASON.

```

Declare the parameters as follows:

```

** Connection handle
01 HCONN          PIC S9(9) BINARY.
** Object handle
01 HOBJ           PIC S9(9) BINARY.
** Message descriptor
01 MSGDESC.
   COPY CMQMDV.
** Options that control the action of MQPUT
01 PUTMSGOPTS.
   COPY CMQPMOV.
** Length of the message in BUFFER
01 BUFFERLENGTH  PIC S9(9) BINARY.
** Message data
01 BUFFER        PIC X(n).
** Completion code
01 COMPCODE       PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON         PIC S9(9) BINARY.

```

PL/I invocation

```

call MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer,
            CompCode, Reason);

```

Declare the parameters as follows:

```

dcl Hconn          fixed bin(31); /* Connection handle */
dcl Hobj           fixed bin(31); /* Object handle */
dcl MsgDesc        like MQMD;     /* Message descriptor */
dcl PutMsgOpts     like MQPMO;     /* Options that control the action of
                                   MQPUT */
dcl BufferLength    fixed bin(31); /* Length of the message in Buffer */
dcl Buffer          char(n);       /* Message data */
dcl CompCode       fixed bin(31); /* Completion code */
dcl Reason         fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```

CALL MQPUT, (HCONN,HOBJ,MSGDESC,PUTMSGOPTS,BUFFERLENGTH, X
            BUFFER,COMPCODE,REASON)

```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HOBJ	DS	F	Object handle
MSGDESC	CMQMDA	,	Message descriptor
PUTMSGOPTS	CMQPMOA	,	Options that control the action of MQPUT
BUFFERLENGTH	DS	F	Length of the message in BUFFER
BUFFER	DS	CL(n)	Message data
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

Visual Basic invocation

```
MQPUT Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason
```

Declare the parameters as follows:

```
Dim Hconn      As Long   'Connection handle'
Dim Hobj       As Long   'Object handle'
Dim MsgDesc    As MQMD   'Message descriptor'
Dim PutMsgOpts As MQPMO   'Options that control the action of MQPUT'
Dim BufferLength As Long  'Length of the message in Buffer'
Dim Buffer      As String 'Message data'
Dim CompCode   As Long   'Completion code'
Dim Reason     As Long   'Reason code qualifying CompCode'
```

MQPUT1 - Put one message

The MQPUT1 call puts one message on a queue, or distribution list, or to a topic.

The queue, distribution list, or topic does not need to be open.

Syntax

MQPUT1 (*Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

ObjDesc

Type: MQOD - input/output

This is a structure that identifies the queue to which the message is added, or the topic to which the message is published. See “MQOD - Object descriptor” on page 455 for details.

If the structure is a queue, the user must be authorized to open the queue for output. The queue must **not** be a model queue.

MsgDesc

Type: MQMD - input/output

This structure describes the attributes of the message being sent, and receives feedback information after the put request is complete. See “MQMD - Message descriptor” on page 396 for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure to specify values for the fields that exist in the version-2 MQMD but not the version-1. Set the *Format* field in the MQMD to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. See “MQMDE - Message descriptor extension” on page 447 for more details.

The application does not need to provide an MQMD structure if a valid message handle is supplied in the *MsgHandle* field of the MQGMO structure or in the *OriginalMsgHandle* or *NewMsgHandle* fields of

the MQPMO structure. If nothing is provided in one of these fields, the descriptor of the message is taken from the descriptor associated with the message handles.

PutMsgOpts

Type: MQPMO - input/output

See “MQPMO - Put-message options” on page 475 for details.

BufferLength

Type: MQLONG - input

The length of the message in *Buffer*. Zero is valid, and indicates that the message contains no application data. The upper limit depends on various factors; see “MQPUT - Put message” on page 720 for the description of the *BufferLength* parameter.

Buffer

Type: MQBYTEExBufferLength - input

This is a buffer containing the application message data to be sent. Align the buffer on a boundary appropriate to the nature of the data in the message. 4-byte alignment is suitable for most messages (including messages containing IBM MQ header structures), but some messages might require more stringent alignment. For example, a message containing a 64-bit binary integer might require 8-byte alignment.

If *Buffer* contains character or numeric data, set the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to the values appropriate to the data; this enables the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All the other parameters on the MQPUT1 call must be in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE).

In the C programming language, the parameter is declared as a pointer-to-void; the address of any type of data can be specified as the parameter.

If the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_MULTIPLE_REASONS

(2136, X'858') Multiple reason codes returned.

MQRC_INCOMPLETE_GROUP

(2241, X'8C1') Message group not complete.

MQRC_INCOMPLETE_MSG

(2242, X'8C2') Logical message not complete.

MQRC_PRIORITY_EXCEEDS_MAXIMUM

(2049, X'801') Message Priority exceeds maximum value supported.

MQRC_UNKNOWN_REPORT_OPTION

(2104, X'838') Report options in message descriptor not recognized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ALIAS_BASE_Q_TYPE_ERROR

(2001, X'7D1') Alias base queue not a valid type.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BACKED_OUT

(2003, X'7D3') Unit of work backed out.

MQRC_BUFFER_ERROR

(2004, X'7D4') Buffer parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') coupling facility not available.

MQRC_CF_STRUC_AUTH_FAILED

(2348, X'92C') Coupling-facility structure authorization check failed.

MQRC_CF_STRUC_ERROR

(2349, X'92D') Coupling-facility structure not valid.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CF_STRUC_LIST_HDR_IN_USE

(2347, X'92B') Coupling-facility structure list-header in use.

MQRC_CFGR_ERROR

(2416, X'970') PCF group parameter structure MQCFGR in the message data is not valid.

MQRC_CFH_ERROR

(2235, X'8BB') PCF header structure not valid.

MQRC_CFIF_ERROR

(2414, X'96E') PCF integer filter parameter structure in the message data is not valid.

MQRC_CFIL_ERROR

(2236, X'8BC') PCF integer list parameter structure or PCIF*64 integer list parameter structure not valid.

MQRC_CFIN_ERROR

(2237, X'8BD') PCF integer parameter structure or PCIF*64 integer parameter structure not valid.

MQRC_CFSF_ERROR

(2415, X'96F') PCF string filter parameter structure in the message data is not valid.

MQRC_CFSL_ERROR

(2238, X'8BE') PCF string list parameter structure not valid.

MQRC_CFST_ERROR

(2239, X'8BF') PCF string parameter structure not valid.

MQRC_CICS_WAIT_FAILED

(2140, X'85C') Wait request rejected by CICS.

MQRC_CLUSTER_EXIT_ERROR

(2266, X'8DA') Cluster workload exit failed.

MQRC_CLUSTER_RESOLUTION_ERROR

(2189, X'88D') Cluster name resolution failed.

MQRC_CLUSTER_RESOURCE_ERROR

(2269, X'8DD') Cluster resource error.

MQRC_COD_NOT_VALID_FOR_XCF_Q

(2106, X'83A') COD report option not valid for XCF queue.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED

(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION QUIESCING

(2202, X'89A') Connection quiescing.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_CONTENT_ERROR

2554 (X'09FA') Message content could not be parsed to determine whether the message can be delivered to a subscriber with an extended message selector.

MQRC_CONTEXT_HANDLE_ERROR

(2097, X'831') Queue handle referred to does not save context.

MQRC_CONTEXT_NOT_AVAILABLE

(2098, X'832') Context not available for queue handle referred to.

MQRC_DATA_LENGTH_ERROR

(2010, X'7DA') Data length parameter not valid.

MQRC_DB2_NOT_AVAILABLE

(2342, X'926') Db2 subsystem not available.

MQRC_DEF_XMIT_Q_TYPE_ERROR

(2198, X'896') Default transmission queue not local.

MQRC_DEF_XMIT_Q_USAGE_ERROR

(2199, X'897') Default transmission queue usage error.

MQRC_DH_ERROR

(2135, X'857') Distribution header structure not valid.

MQRC_DLH_ERROR

(2141, X'85D') Dead letter header structure not valid.

MQRC_EPH_ERROR

(2420, X'974') Embedded PCF structure not valid.

MQRC_EXPIRY_ERROR

(2013, X'7DD') Expiry time not valid.

MQRC_FEEDBACK_ERROR
(2014, X'7DE') Feedback code not valid.

MQRC_GLOBAL_UOW_CONFLICT
(2351, X'92F') Global units of work conflict.

MQRC_GROUP_ID_ERROR
(2258, X'8D2') Group identifier not valid.

MQRC_HANDLE_IN_USE_FOR_UOW
(2353, X'931') Handle in use for global unit of work.

MQRC_HANDLE_NOT_AVAILABLE
(2017, X'7E1') No more handles available.

MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQRC_HEADER_ERROR
(2142, X'85E') IBM MQ header structure not valid.

MQRC_IIH_ERROR
(2148, X'864') IMS information header structure not valid.

MQRC_LOCAL_UOW_CONFLICT
(2352, X'930') Global unit of work conflicts with local unit of work.

MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.

MQRC_MDE_ERROR
(2248, X'8C8') Message descriptor extension not valid.

MQRC_MISSING_REPLY_TO_Q
(2027, X'7EB') Missing reply-to queue.

MQRC_MISSING_WIH
(2332, X'91C') Message data does not begin with MQWIH.

MQRC_MSG_FLAGS_ERROR
(2249, X'8C9') Message flags not valid.

MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.

MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.

MQRC_MSG_TOO_BIG_FOR_Q_MGR
(2031, X'7EF') Message length greater than maximum for queue manager.

MQRC_MSG_TYPE_ERROR
(2029, X'7ED') Message type in message descriptor not valid.

MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.

MQRC_NO_DESTINATIONS_AVAILABLE
(2270, X'8DE') No destination queues available.

MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.

MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQRC_OBJECT_IN_USE
(2042, X'7FA') Object already open with conflicting options.

MQRC_OBJECT_LEVEL_INCOMPATIBLE
(2360, X'938') Object level not compatible.

MQRC_OBJECT_NAME_ERROR
(2152, X'868') Object name not valid.

MQRC_OBJECT_NOT_UNIQUE
(2343, X'927') Object not unique.

MQRC_OBJECT_Q_MGR_NAME_ERROR
(2153, X'869') Object queue-manager name not valid.

MQRC_OBJECT_RECORDS_ERROR
(2155, X'86B') Object records not valid.

MQRC_OBJECT_TYPE_ERROR
(2043, X'7FB') Object type not valid.

MQRC_OD_ERROR
(2044, X'7FC') Object descriptor structure not valid.

MQRC_OFFSET_ERROR
(2251, X'8CB') Message segment offset not valid.

MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.

MQRC_ORIGINAL_LENGTH_ERROR
(2252, X'8CC') Original length not valid.

MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page-set data set.

MQRC_PAGESET_FULL
(2192, X'890') External storage medium is full.

MQRC_PCF_ERROR
(2149, X'865') PCF structures not valid.

MQRC_PERSISTENCE_ERROR
(2047, X'7FF') Persistence not valid.

MQRC_PERSISTENT_NOT_ALLOWED
(2048, X'800') Queue does not support persistent messages.

MQRC_PMO_ERROR
(2173, X'87D') Put-message options structure not valid.

MQRC_PMO_RECORD_FLAGS_ERROR
(2158, X'86E') Put message record flags not valid.

MQRC_PRIORITY_ERROR
(2050, X'802') Message priority not valid.

MQRC_PUBLICATION_FAILURE
(2502, X'9C6') The publication has not been delivered to any of the subscribers.

MQRC_PUT_INHIBITED
(2051, X'803') Put calls inhibited for the queue.

MQRC_PUT_MSG_RECORDS_ERROR
(2159, X'86F') Put message records not valid.

MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.

MQRC_Q_FULL
(2053, X'805') Queue already contains maximum number of messages.

MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.

MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.

MQRC_Q_SPACE_NOT_AVAILABLE

(2056, X'808') No space available on disk for queue.

MQRC_Q_TYPE_ERROR

(2057, X'809') Queue type not valid.

MQRC_RECS_PRESENT_ERROR

(2154, X'86A') Number of records present not valid.

MQRC_REMOTE_Q_NAME_ERROR

(2184, X'888') Remote queue name not valid.

MQRC_REPORT_OPTIONS_ERROR

(2061, X'80D') Report options in message descriptor not valid.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_RESPONSE_RECORDS_ERROR

(2156, X'86C') Response records not valid.

MQRC_RFH_ERROR

(2334, X'91E') MQRFH or MQRFH2 structure not valid.

MQRC_RMH_ERROR

(2220, X'8AC') Reference message header structure not valid.

MQRC_SECURITY_ERROR

(2063, X'80F') Security error occurred.

MQRC_SEGMENT_LENGTH_ZERO

(2253, X'8CD') Length of data in message segment is zero.

MQRC_SELECTION_NOT_AVAILABLE

2551 (X'09F7') A possible subscriber for the publication exists, but the queue manager cannot check whether to send the publication to the subscriber.

MQRC_STOPPED_BY_CLUSTER_EXIT

(2188, X'88C') Call rejected by cluster workload exit.

MQRC_STORAGE_CLASS_ERROR

(2105, X'839') Storage class error.

MQRC_STORAGE_MEDIUM_FULL

(2192, X'890') External storage medium is full.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT

(2109, X'83D') Call suppressed by exit program.

MQRC_SYNCPOINT_LIMIT_REACHED

(2024, X'7E8') No more messages can be handled within current unit of work.

MQRC_SYNCPOINT_NOT_AVAILABLE

(2072, X'818') Syncpoint support not available.

MQRC_TM_ERROR

(2265, X'8D9') Trigger message structure not valid.

MQRC_TMC_ERROR

(2191, X'88F') Character trigger message structure not valid.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

MQRC_UNKNOWN_ALIAS_BASE_Q

(2082, X'822') Unknown alias base queue.

MQRC_UNKNOWN_DEF_XMIT_Q

(2197, X'895') Unknown default transmission queue.

MQRC_UNKNOWN_OBJECT_NAME

(2085, X'825') Unknown object name.

MQRC_UNKNOWN_OBJECT_Q_MGR

(2086, X'826') Unknown object queue manager.

MQRC_UNKNOWN_REMOTE_Q_MGR

(2087, X'827') Unknown remote queue manager.

MQRC_UNKNOWN_XMIT_Q

(2196, X'894') Unknown transmission queue.

MQRC_UOW_ENLISTMENT_ERROR

(2354, X'932') Enlistment in global unit of work failed.

MQRC_UOW_MIX_NOT_SUPPORTED

(2355, X'933') Mixture of unit-of-work calls not supported.

MQRC_UOW_NOT_AVAILABLE

(2255, X'8CF') Unit of work not available for the queue manager to use.

MQRC_WIH_ERROR

(2333, X'91D') MQWIH structure not valid.

MQRC_WRONG_CF_LEVEL

(2366, X'93E') Coupling-facility structure is wrong level.

MQRC_WRONG_MD_VERSION

(2257, X'8D1') Wrong version of MQMD supplied.

MQRC_XMIT_Q_TYPE_ERROR

(2091, X'82B') Transmission queue not local.

MQRC_XMIT_Q_USAGE_ERROR

(2092, X'82C') Transmission queue with wrong usage.

MQRC_XQH_ERROR

(2260, X'8D4') Transmission queue header structure not valid.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

- Both the MQPUT and MQPUT1 calls can be used to put messages on a queue; which call to use depends on the circumstances:
 - Use the MQPUT call to place multiple messages on the *same* queue.
An MQOPEN call specifying the MQOO_OUTPUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.
 - Use the MQPUT1 call to put only *one* message on a queue.
This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, minimizing the number of calls that must be issued.
- If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if certain conditions are satisfied. However, in most environments the MQPUT1 call does not satisfy these conditions, and so does not preserve message order. The MQPUT call must be used instead in these environments. See [MQPUT usage notes](#) for details.
- The MQPUT1 call can be used to put messages to distribution lists. For general information about this, see the usage notes for the MQOPEN and MQPUT calls.

Distribution lists are supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

The following differences apply when using the MQPUT1 call:

- a. If the application provides MQRR response records, they must be provided using the MQOD structure; they cannot be provided using the MQPMO structure.
- b. The reason code MQRC_OPEN_FAILED is never returned by MQPUT1 in the response records; if a queue fails to open, the response record for that queue contains the reason code resulting from the open operation.

If an open operation for a queue succeeds with a completion code of MQCC_WARNING, the completion code and reason code in the response record for that queue are replaced by the completion and reason codes resulting from the put operation.

As with the MQOPEN and MQPUT calls, the queue manager sets the response records (if provided) only when the outcome of the call is not the same for all queues in the distribution list; this is indicated by the call completing with reason code MQRC_MULTIPLE_REASONS.

4. If the MQPUT1 call is used to put a message on a cluster queue, the call behaves as though MQOO_BIND_NOT_FIXED had been specified on the MQOPEN call.
5. If a message is put with one or more IBM MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. For more information about this, see the usage notes for the MQPUT call.
6. If more than one of the warning situations arise (see the *CompCode* parameter), the reason code returned is the *first* one in the following list that applies:
 - a. MQRC_MULTIPLE_REASONS
 - b. MQRC_INCOMPLETE_MSG
 - c. MQRC_INCOMPLETE_GROUP
 - d. MQRC_PRIORITY_EXCEEDS_MAXIMUM or MQRC_UNKNOWN_REPORT_OPTION
7. For the Visual Basic programming language, the following points apply:
 - If the size of the *Buffer* parameter is less than the length specified by the *BufferLength* parameter, the call fails with reason code MQRC_BUFFER_LENGTH_ERROR.
 - The *Buffer* parameter is declared as being of type String. If the data to be placed on the queue is not of type String, use the MQPUT1Any call in place of MQPUT1.

The MQPUT1Any call has the same parameters as the MQPUT1 call, except that the *Buffer* parameter is declared as being of type Any, allowing any type of data to be placed on the queue. However, this means that *Buffer* cannot be checked to ensure that it is at least *BufferLength* bytes in size.
8. When an MQPUT1 call is issued with MQPMO_SYNCPOINT, the default behavior changes, so that the put operation is completed asynchronously. This might cause a change in the behavior of some applications that rely on certain fields in the MQOD and MQMD structures being returned, but which now contain undefined values. An application can specify MQPMO_SYNC_RESPONSE to ensure that the put operation is performed synchronously and that all the appropriate field values are completed.

C invocation

```
MQPUT1 (Hconn, &ObjDesc, &MsgDesc, &PutMsgOpts,
        BufferLength, Buffer, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */
MQOD      ObjDesc;         /* Object descriptor */
MQMD      MsgDesc;        /* Message descriptor */
MQPMO     PutMsgOpts;     /* Options that control the action of MQPUT1 */
MQLONG    BufferLength;    /* Length of the message in Buffer */
MQBYTE    Buffer[n];       /* Message data */
MQLONG    CompCode;       /* Completion code */
MQLONG    Reason;         /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQPUT1' USING HCONN, OBJDESC, MSGDESC, PUTMSGOPTS,  
                   BUFFERLENGTH, BUFFER, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN      PIC S9(9) BINARY.  
** Object descriptor  
01 OBJDESC.  
   COPY CMQODV.  
** Message descriptor  
01 MSGDESC.  
   COPY CMQMDV.  
** Options that control the action of MQPUT1  
01 PUTMSGOPTS.  
   COPY CMQPMOV.  
** Length of the message in BUFFER  
01 BUFFERLENGTH PIC S9(9) BINARY.  
** Message data  
01 BUFFER      PIC X(n).  
** Completion code  
01 COMPCODE    PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON      PIC S9(9) BINARY.
```

PL/I invocation

```
call MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer,  
            CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */  
dcl ObjDesc    like MQOD;    /* Object descriptor */  
dcl MsgDesc    like MQMD;    /* Message descriptor */  
dcl PutMsgOpts like MQPMO;    /* Options that control the action of  
                               MQPUT1 */  
dcl BufferLength fixed bin(31); /* Length of the message in Buffer */  
dcl Buffer      char(n);      /* Message data */  
dcl CompCode   fixed bin(31); /* Completion code */  
dcl Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQPUT1,(HCONN,OBJDESC,MSGDESC,PUTMSGOPTS,BUFFERLENGTH,    X  
            BUFFER,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
OBJDESC	CMQODA	,	Object descriptor
MSGDESC	CMQMDA	,	Message descriptor
PUTMSGOPTS	CMQPMOA	,	Options that control the action of MQPUT1
BUFFERLENGTH	DS	F	Length of the message in BUFFER
BUFFER	DS	CL(n)	Message data
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

Visual Basic invocation

```
MQPUT1 Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer,  
      CompCode, Reason
```

Declare the parameters as follows:

```

Dim Hconn      As Long 'Connection handle'
Dim ObjDesc    As MQOD  'Object descriptor'
Dim MsgDesc    As MQMD  'Message descriptor'
Dim PutMsgOpts As MQPMO 'Options that control the action of MQPUT1'
Dim BufferLength As Long 'Length of the message in Buffer'
Dim Buffer      As String 'Message data'
Dim CompCode   As Long  'Completion code'
Dim Reason     As Long  'Reason code qualifying CompCode'

```

MQSET - Set object attributes

Use the MQSET call to change the attributes of an object represented by a handle. The object must be a queue.

Syntax

MQSET (*Hconn*, *Hobj*, *SelectorCount*, *Selectors*, *IntAttrCount*, *IntAttrs*, *CharAttrLength*, *CharAttrs*, *Compcode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Hobj

Type: MQHOBJ - input

This handle represents the queue object with attributes that are to be set. The handle was returned by a previous MQOPEN call that specified the MQOO_SET option.

SelectorCount

Type: MQLONG - input

This is the count of selectors that are supplied in the *Selectors* array. It is the number of attributes that are to be set. Zero is a valid value. The maximum number allowed is 256.

Selectors

Type: MQLONGxSelectorCount - input

This is an array of *SelectorCount* attribute selectors; each selector identifies an attribute (integer or character) with a value that is to be set.

Each selector must be valid for the type of queue that *Hobj* represents. Only certain MQIA_* and MQCA_* values are allowed; as listed later.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (MQIA_* selectors) must be specified in *IntAttrs* in the same order in which these selectors occur in *Selectors*. Attribute values that correspond to character attribute selectors (MQCA_* selectors) must be specified in *CharAttrs* in the same order in which those selectors occur. MQIA_* selectors can be interleaved with the MQCA_* selectors; only the relative order within each type is important.

You can specify the same selector more than once; if you do, the last value specified for a particular selector is the one that takes effect.

Note:

1. The integer and character attribute selectors are allocated within two different ranges; the MQIA_* selectors reside within the range MQIA_FIRST through MQIA_LAST, and the MQCA_* selectors within the range MQCA_FIRST through MQCA_LAST.

For each range, the constants MQIA_LAST_USED and MQCA_LAST_USED define the highest value that the queue manager accepts.

2. If all the MQIA_* selectors occur first, the same element numbers can be used to address corresponding elements in the *Selectors* and *IntAttrs* arrays.
3. If the *SelectorCount* parameter is zero, *Selectors* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler might be null.

The attributes that can be set are listed in the following table. No other attributes can be set using this call. For the MQCA_* attribute selectors, the constant that defines the length in bytes of the string that is required in *CharAttrs* is supplied in parentheses.

Table 559. MQSET attribute selectors for queues		
Selector	Description	Note
MQCA_TRIGGER_DATA	Trigger data (MQ_TRIGGER_DATA_LENGTH).	
MQIA_DIST_LISTS	Distribution list support.	1
MQIA_INHIBIT_GET	Whether get operations are allowed.	
MQIA_INHIBIT_PUT	Whether put operations are allowed.	
MQIA_TRIGGER_CONTROL	Trigger control.	
MQIA_TRIGGER_DEPTH	Trigger depth.	
MQIA_TRIGGER_MSG_PRIORITY	Threshold message priority for triggers.	
MQIA_TRIGGER_TYPE	Trigger type.	

Note:

1. Supported only on AIX, HP-UX, IBM i, Solaris, Linux, Windows, plus IBM MQ MQI clients connected to these systems.

IntAttrCount

Type: MQLONG - input

This is the number of elements in the *IntAttrs* array, and must be at least the number of MQIA_* selectors in the *Selectors* parameter. Zero is a valid value if there are none.

IntAttrs

Type: MQLONGxIntAttrCount - input

This is an array of *IntAttrCount* integer attribute values. These attribute values must be in the same order as the MQIA_* selectors in the *Selectors* array.

If the *IntAttrCount* or *SelectorCount* parameter is zero, *IntAttrs* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler might be null.

CharAttrLength

Type: MQLONG - input

This is the length in bytes of the *CharAttrs* parameter, and must be at least the sum of the lengths of the character attributes specified in the *Selectors* array. Zero is a valid value if there are no MQCA_* selectors in *Selectors*.

CharAttrs

Type: MQCHARxCharAttrLength - input

This is the buffer containing the character attribute values, concatenated together. The length of the buffer is given by the *CharAttrLength* parameter.

The characters attributes must be specified in the same order as the MQCA_* selectors in the *Selectors* array. The length of each character attribute is fixed (see *Selectors*). If the value to be set for an attribute contains fewer nonblank characters than the defined length of the attribute, pad the value in *CharAttrs* to the right with blanks to make the attribute value match the defined length of the attribute.

If the *CharAttrLength* or *SelectorCount* parameter is zero, *CharAttrs* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler might be null.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed.

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CF_NOT_AVAILABLE

(2345, X'929') Coupling facility not available.

MQRC_CF_STRUC_FAILED

(2373, X'945') Coupling-facility structure failed.

MQRC_CF_STRUC_IN_USE

(2346, X'92A') Coupling-facility structure in use.

MQRC_CF_STRUC_LIST_HDR_IN_USE

(2347, X'92B') Coupling-facility structure list-header in use.

MQRC_CHAR_ATTR_LENGTH_ERROR

(2006, X'7D6') Length of character attributes not valid.

MQRC_CHAR_ATTRS_ERROR
(2007, X'7D7') Character attributes string not valid.

MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.

MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.

MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.

MQRC_DB2_NOT_AVAILABLE
(2342, X'926') Db2 subsystem not available.

MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.

MQRC_INHIBIT_VALUE_ERROR
(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.

MQRC_INT_ATTR_COUNT_ERROR
(2021, X'7E5') Count of integer attributes not valid.

MQRC_INT_ATTRS_ARRAY_ERROR
(2023, X'7E7') Integer attributes array not valid.

MQRC_NOT_OPEN_FOR_SET
(2040, X'7F8') Queue not open for set.

MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.

MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page-set data set.

MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.

MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.

MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.

MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.

MQRC_SELECTOR_COUNT_ERROR
(2065, X'811') Count of selectors not valid.

MQRC_SELECTOR_ERROR
(2067, X'813') Attribute selector not valid.

MQRC_SELECTOR_LIMIT_EXCEEDED
(2066, X'812') Count of selectors too large.

MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.

MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.

MQRC_TRIGGER_CONTROL_ERROR

(2075, X'81B') Value for trigger-control attribute not valid.

MQRC_TRIGGER_DEPTH_ERROR

(2076, X'81C') Value for trigger-depth attribute not valid.

MQRC_TRIGGER_MSG_PRIORITY_ERR

(2077, X'81D') Value for trigger-message-priority attribute not valid.

MQRC_TRIGGER_TYPE_ERROR

(2078, X'81E') Value for trigger-type attribute not valid.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. Using this call, the application can specify an array of integer attributes, or a collection of character attribute strings, or both. If no errors occur, the attributes specified are all set simultaneously. If an error occurs (for example, if a selector is not valid, or an attempt is made to set an attribute to a value that is not valid), the call fails and no attributes are set.
2. The values of attributes can be determined using the MQINQ call; see [“MQINQ - Inquire object attributes”](#) on page 677 for details.

Note: Not all attributes with values that can be inquired using the MQINQ call can have their values changed using the MQSET call. For example, no process-object or queue-manager attributes can be set with this call.

3. Attribute changes are preserved across restarts of the queue manager (other than alterations to temporary dynamic queues, which do not survive restarts of the queue manager).
4. You cannot change the attributes of a model queue using the MQSET call. However, if you open a model queue using the MQOPEN call with the MQOO_SET option, you can use the MQSET call to set the attributes of the dynamic local queue that is created by the MQOPEN call.
5. If the object being set is a cluster queue, there must be a local instance of the cluster queue for the open to succeed.

For more information about object attributes, see:

- [“Attributes for queues”](#) on page 801
- [“Attributes for namelists”](#) on page 832
- [“Attributes for process definitions”](#) on page 834
- [“Attributes for the queue manager”](#) on page 768

C invocation

```
MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
       CharAttrLength, CharAttrs, &CompCode, &Reason);
```

Declare the parameters as follows:

```

MQHCONN  Hconn;           /* Connection handle */
MQHOBJ   Hobj;            /* Object handle */
MQLONG   SelectorCount;   /* Count of selectors */
MQLONG   Selectors[n];    /* Array of attribute selectors */
MQLONG   IntAttrCount;    /* Count of integer attributes */
MQLONG   IntAttrs[n];     /* Array of integer attributes */
MQLONG   CharAttrLength;  /* Length of character attributes buffer */
MQCHAR   CharAttrs[n];    /* Character attributes */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying CompCode */

```

COBOL invocation

```
CALL 'MQSET' USING HCONN, HOBJ, SELECTORCOUNT, SELECTORS-TABLE,  
                  INTATTRCOUNT, INTATTRS-TABLE, CHARATTRLENGTH,  
                  CHARATTRS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN          PIC S9(9) BINARY.  
** Object handle  
01 HOBJ          PIC S9(9) BINARY.  
** Count of selectors  
01 SELECTORCOUNT PIC S9(9) BINARY.  
** Array of attribute selectors  
01 SELECTORS-TABLE.  
02 SELECTORS     PIC S9(9) BINARY OCCURS n TIMES.  
** Count of integer attributes  
01 INTATTRCOUNT PIC S9(9) BINARY.  
** Array of integer attributes  
01 INTATTRS-TABLE.  
02 INTATTRS     PIC S9(9) BINARY OCCURS n TIMES.  
** Length of character attributes buffer  
01 CHARATTRLENGTH PIC S9(9) BINARY.  
** Character attributes  
01 CHARATTRS     PIC X(n).  
** Completion code  
01 COMPCODE      PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON        PIC S9(9) BINARY.
```

PL/I invocation

```
call MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,  
            IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn          fixed bin(31); /* Connection handle */  
dcl Hobj          fixed bin(31); /* Object handle */  
dcl SelectorCount  fixed bin(31); /* Count of selectors */  
dcl Selectors(n)   fixed bin(31); /* Array of attribute selectors */  
dcl IntAttrCount   fixed bin(31); /* Count of integer attributes */  
dcl IntAttrs(n)    fixed bin(31); /* Array of integer attributes */  
dcl CharAttrLength fixed bin(31); /* Length of character attributes  
                                buffer */  
dcl CharAttrs      char(n);      /* Character attributes */  
dcl CompCode       fixed bin(31); /* Completion code */  
dcl Reason         fixed bin(31); /* Reason code qualifying  
                                CompCode */
```

High Level Assembler invocation

```
CALL MQSET, (HCONN,HOBJ,SELECTORCOUNT,SELECTORS,INTATTRCOUNT, X  
            INTATTRS,CHARATTRLENGTH,CHARATTRS,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HOBJ	DS	F	Object handle
SELECTORCOUNT	DS	F	Count of selectors
SELECTORS	DS	(n)F	Array of attribute selectors
INTATTRCOUNT	DS	F	Count of integer attributes
INTATTRS	DS	(n)F	Array of integer attributes
CHARATTRLENGTH	DS	F	Length of character attributes buffer
CHARATTRS	DS	CL(n)	Character attributes
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

Visual Basic invocation

```
MQSET Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,  
CharAttrLength, CharAttrs, CompCode, Reason
```

Declare the parameters as follows:

```
Dim Hconn          As Long   'Connection handle'  
Dim Hobj           As Long   'Object handle'  
Dim SelectorCount  As Long   'Count of selectors'  
Dim Selectors      As Long   'Array of attribute selectors'  
Dim IntAttrCount   As Long   'Count of integer attributes'  
Dim IntAttrs       As Long   'Array of integer attributes'  
Dim CharAttrLength As Long   'Length of character attributes buffer'  
Dim CharAttrs      As String 'Character attributes'  
Dim CompCode       As Long   'Completion code'  
Dim Reason         As Long   'Reason code qualifying CompCode'
```

MQSETMP - Set message property

Use the MQSETMP call to set or modify a property of a message handle.

Syntax

MQSETMP (*Hconn, Hmsg, SetPropOpts, Name, PropDesc, Type, ValueLength, Value, Compcode, Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager.

The value must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter. If the message handle was created using MQHC_UNASSOCIATED_HCONN, a valid connection must be established on the thread setting a property of the message handle, otherwise the call fails with reason code MQRC_CONNECTION_BROKEN.

Hmsg

Type: MQHMSG - input

This is the message handle to be modified. The value was returned by a previous MQCRTMH call.

SetPropOpts

Type: MQSMPO - input

Control how message properties are set.

This structure allows applications to specify options that control how message properties are set. The structure is an input parameter on the MQSETMP call. See [MQSMPO](#) for further information.

Name

Type: MQCHARV- input

This is the name of the property to set.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

PropDesc

Type: MQPD - input/output

This structure is used to define the attributes of a property, including:

- what happens if the property is not supported
- what message context the property belongs to
- what messages the property is copied into as it flows

See [MQPD](#) for further information about this structure.

Type

Type: MQLONG - input

The data type of the property being set. It can be one of the following:

MQTYPE_BOOLEAN

A Boolean. *ValueLength* must be 4.

MQTYPE_BYTE_STRING

A byte string. *ValueLength* must be zero or greater.

MQTYPE_INT8

An 8-bit signed integer. *ValueLength* must be 1.

MQTYPE_INT16

A 16-bit signed integer. *ValueLength* must be 2.

MQTYPE_INT32

A 32-bit signed integer. *ValueLength* must be 4.

MQTYPE_INT64

A 64-bit signed integer. *ValueLength* must be 8.

MQTYPE_FLOAT32

A 32-bit floating-point number. *ValueLength* must be 4.

Note: this type is not supported with applications using IBM COBOL for z/OS.

MQTYPE_FLOAT64

A 64-bit floating-point number. *ValueLength* must be 8.

Note: this type is not supported with applications using IBM COBOL for z/OS.

MQTYPE_STRING

A character string. *ValueLength* must be zero or greater, or the special value MQVL_NULL_TERMINATED.

MQTYPE_NULL

The property exists but has a null value. *ValueLength* must be zero.

ValueLength

Type: MQLONG - input

The length in bytes of the property value in the *Value* parameter. Zero is valid only for null values or for strings or byte strings. Zero indicates that the property exists but that the value contains no characters or bytes.

The value must be greater than or equal to zero or the following special value if the *Type* parameter has MQTYPE_STRING set:

MQVL_NULL_TERMINATED

The value is delimited by the first null encountered in the string. The null is not included as part of the string. This value is invalid if MQTYPE_STRING is not also set.

Note: The null character used to terminate a string if MQVL_NULL_TERMINATED is set is a null from the character set of the Value.

Value

Type: MQBYTEExValueLength - input

The value of the property to be set. The buffer must be aligned on a boundary appropriate to the nature of the data in the value.

In the C programming language, the parameter is declared as a pointer-to-void; the address of any type of data can be specified as the parameter.

If *ValueLength* is zero, *Value* is not referred to. In this case, the parameter address passed by programs written in C or System/390 assembler can be null.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_RFH_FORMAT_ERROR

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'089C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR

(2130, X'852') Unable to load adapter service module.

MQRC_ASID_MISMATCH

(2157, X'86D') Primary and home ASIDs differ.

MQRC_BUFFER_ERROR

(2004, X'07D4') Value parameter not valid.

MQRC_BUFFER_LENGTH_ERROR

(2005, X'07D5') Value length parameter not valid.

MQRC_CALL_IN_PROGRESS

(2219, X'08AB') MQI call entered before previous call completed.

MQRC_HMSG_ERROR

(2460, X'099C') Message handle pointer not valid.

MQRC_MSG_HANDLE_IN_USE

(2499, X'09C3') Message handle already in use.

MQRC_OPTIONS_ERROR

(2046, X'07FE') Options not valid or not consistent.

MQRC_PD_ERROR

(2482, X'09B2') Property descriptor structure not valid.

MQRC_PROPERTY_NAME_ERROR

(2442, X'098A') Invalid property name.

MQRC_PROPERTY_TYPE_ERROR

(2473, X'09A9') Invalid property data type.

MQRC_PROP_NUMBER_FORMAT_ERROR

(2472, X'09A8') Number format error encountered in value data.

MQRC_SMPO_ERROR

(2463, X'099F') Set message property options structure not valid.

MQRC_SOURCE_CCSID_ERROR

(2111, X'083F') Property name coded character set identifier not valid.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C invocation

```
MQSETMP (Hconn, Hmsg, &SetPropOpts, &Name, &PropDesc, Type,  
ValueLength, &Value, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;          /* Connection handle */  
MQHMSG   Hmsg;           /* Message handle */  
MQSMPO   SetPropOpts;    /* Options that control the action of MQSETMP */  
MQCHARV  Name;           /* Property name */  
MQPD     PropDesc;       /* Property descriptor */  
MQLONG   Type;           /* Property data type */  
MQLONG   ValueLength;    /* Length of property value in Value */  
MQBYTE   Value[n];       /* Property value */  
MQLONG   CompCode;       /* Completion code */  
MQLONG   Reason;        /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQSETMP' USING HCONN, HMSG, SETMSGOPTS, NAME, PROPDSC, TYPE,  
                     VALUELENGTH, VALUE, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN PIC S9(9) BINARY.  
** Message handle  
01 HMSG PIC S9(18) BINARY.  
** Options that control the action of MQSETMP  
01 SETMSGOPTS.  
   COPY CMQSMPOV.  
** Property name  
01 NAME  
   COPY CMQCHRVV.  
** Property descriptor  
01 PROPDSC.  
   COPY CMQPDV.  
** Property data type  
01 TYPE PIC S9(9) BINARY.  
** Length of property value in VALUE  
01 VALUELENGTH PIC S9(9) BINARY.  
** Property value  
01 VALUE PIC X(n).  
** Completion code  
01 COMPCODE PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON PIC S9(9) BINARY.
```

PL/I invocation

```
call MQSETMP (Hconn, Hmsg, SetPropOpts, Name, PropDesc, Type, ValueLength,  
              Value, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn      fixed bin(31); /* Connection handle */  
dcl Hmsg       fixed bin(63); /* Message handle */  
dcl SetPropOpts like MQSMPO;  /* Options that control the action of MQSETMP */  
dcl Name       like MQCHARV;  /* Property name */  
dcl PropDesc   like MQPD;     /* Property descriptor */  
dcl Type       fixed bin(31); /* Property data type */
```

```

dcl ValueLength fixed bin(31); /* Length of property value in Value */
dcl Value       char(n);      /* Property value */
dcl CompCode    fixed bin(31); /* Completion code */
dcl Reason      fixed bin(31); /* Reason code qualifying CompCode */

```

High Level Assembler invocation

```

CALL MQSETMP, (HCONN, HMSG, SETMSGHOPTS, NAME, PROPDSC, TYPE, VALUELENGTH,
              VALUE, COMPCODE, REASON)

```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
HMSG	DS	D	Message handle
SETMSGHOPTS	CMQSMPOA	,	Options that control the action of MQSETMP
NAME	CMQCHRVA	,	Property name
PROPDSC	CMQPDA	,	Property descriptor
TYPE	DS	F	Property data type
VALUELENGTH	DS	F	Length of property value in VALUE
VALUE	DS	CL(n)	Property value
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQSTAT - Retrieve status information

Use the MQSTAT call to retrieve status information. The type of status information returned is determined by the Type value specified on the call.

Syntax

MQSTAT (*Hconn*, *Type*, *Stat*, *Compcode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Type

Type: MQLONG - input

Type of status information being requested. The valid values are:

MQSTAT_TYPE_ASYNC_ERROR

Return information about previous asynchronous put operations.

MQSTAT_TYPE_RECONNECTION

Return information about reconnection. If the connection is reconnecting or failed to reconnect, the information describes the failure which caused the connection to begin reconnecting.

This value is only valid for client connections. For other types of connection, the call fails with reason code **MQRC_ENVIRONMENT_ERROR**

MQSTAT_TYPE_RECONNECTION_ERROR

Return information about a previous failure related to reconnect. If the connection failed to reconnect, the information describes the failure which caused reconnection to fail.

This value is only valid for client connections. For other types of connection, the call fails with reason code **MQRC_ENVIRONMENT_ERROR**.

Stat

Type: MQSTS - input/output

Status information structure. See [“MQSTS - Status reporting structure”](#) on page 564 for details.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_API_EXIT_ERROR

(2374, X'946') API exit failed

MQRC_API_EXIT_LOAD_ERROR

(2183, X'887') Unable to load API exit.

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete.

MQRC_CONNECTION_BROKEN

(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down.

MQRC_FUNCTION_NOT_SUPPORTED

(2298, X'8FA') The function requested is not available in the current environment.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_Q_MGR_STOPPING

(2162, X'872') - Queue manager stopping

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_STAT_TYPE_ERROR

(2430, X'97E') Error with MQSTAT type

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_STS_ERROR

(2426, X'97A') Error with MQSTS structure

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

1. A call to MQSTAT specifying a type of MQSTAT_TYPE_ASYNC_ERROR returns information about previous asynchronous MQPUT and MQPUT1 operations. The MQSTS structure passed back on return from the MQSTAT call contains the first recorded asynchronous warning or error information for that connection. If further errors or warnings follow the first, they do not normally alter these values. However, if an error occurs with a completion code of MQCC_WARNING, a subsequent failure with a completion code of MQCC_FAILED is returned instead.
2. If no errors have occurred since the connection was established or since the last call to MQSTAT then a CompCode of MQCC_OK and Reason of MQRC_NONE are returned in the MQSTS structure.
3. Counts of the number of asynchronous calls that have been processed under the connection handle are returned by way of three counter fields; PutSuccessCount, PutWarningCount and PutFailureCount. These counters are incremented by the queue manager each time an asynchronous operation is processed successfully, has a warning, or fails (note that for accounting purposes a put to a distribution list counts once per destination queue rather than once per distribution list). A counter is not incremented beyond the maximum positive value AMQ_LONG_MAX.
4. A successful call to MQSTAT results in any previous error information or counts being reset.
5. The behavior of MQSTAT depends on the value of the MQSTAT Type parameter you provide.
6. **MQSTAT_TYPE_ASYNC_ERROR**
 - a. A call to MQSTAT specifying a type of MQSTAT_TYPE_ASYNC_ERROR returns information about previous asynchronous MQPUT and MQPUT1 operations. The MQSTS structure passed back on return from the MQSTAT call contains the first recorded asynchronous warning or error information for that connection. If further errors or warnings follow the first, they do not normally alter these values. However, if an error occurs with a completion code of MQCC_WARNING, a subsequent failure with a completion code of MQCC_FAILED is returned instead.
 - b. If no errors have occurred since the connection was established or since the last call to MQSTAT then a CompCode of MQCC_OK and Reason of MQRC_NONE are returned in the MQSTS structure.
 - c. Counts of the number of asynchronous calls that have been processed under the connection handle are returned by way of three counter fields; PutSuccessCount, PutWarningCount and PutFailureCount. These counters are incremented by the queue manager each time an asynchronous operation is processed successfully, has a warning, or fails (note that for accounting purposes a put to a distribution list counts once per destination queue rather than once per distribution list). A counter is not incremented beyond the maximum positive value AMQ_LONG_MAX.
 - d. A successful call to MQSTAT results in any previous error information or counts being reset.

MQSTAT_TYPE_RECONNECTION

Suppose you call MQSTAT with Type set to MQSTAT_TYPE_RECONNECTION inside an event handler during reconnection. Consider these examples.

The client is attempting reconnection or failed to reconnect.

CompCode in the MQSTS structure is MQCC_FAILED and Reason might be either MQRC_CONNECTION_BROKEN or MQRC_Q_MGR QUIESCING. ObjectType is MQOT_Q_MGR, ObjectName is the name of the queue manager, and ObjectQMgrName is blank.

The client completed reconnection successfully or was never disconnected.

CompCode in the MQSTS structure is MQCC_OK and the Reason is MQRC_NONE

Subsequent calls to MQSTAT return the same results.

MQSTAT_TYPE_RECONNECTION_ERROR

Suppose you call MQSTAT with Type set to MQSTAT_TYPE_RECONNECTION_ERROR in response to receiving MQRC_RECONNECT_FAILED to an MQI call. Consider these examples.

An authorization failure occurred when a queue was being reopened during reconnection to a different queue manager.

CompCode in the MQSTS structure is MQCC_FAILED and Reason is the reason that the reconnection failed, such as MQRC_NOT_AUTHORIZED. ObjectType is the type of object that caused the problem, such as MQOT_QUEUE, ObjectName is the name of the queue and ObjectQMgrName the name of the queue manager owning the queue.

A socket connection error occurred during reconnection.

CompCode in the MQSTS structure is MQCC_FAILED and Reason is the reason that the reconnection failed, such as MQRC_HOST_NOT_AVAILABLE. ObjectType is MQOT_Q_MGR, ObjectName is the name of the queue manager, and ObjectQMgrName is blank.

Subsequent calls to MQSTAT return the same results.

C invocation

```
MQSTAT (Hconn, StatType, &Stat, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN Hconn;          /* Connection Handle */
MQLONG StatType;        /* Status type */
MQSTS Stat;             /* Status information structure */
MQLONG CompCode;        /* Completion code */
MQLONG Reason;          /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQSTAT' USING HCONN, STATTYPE, STAT, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**      Connection handle
01      HCONN      PIC S9(9)          BINARY.
**      Status type
01      STATTYPE   PIC S9(9)          BINARY.
**      Status information
01      STAT.
      COPY CMQSTSV.
**      Completion code
01      COMPCODE   PIC S9(9)          BINARY.
**      Reason code qualifying COMPCODE
01      REASON     PIC S9(9)          BINARY.
```

PL/I invocation

```
call MQSTAT (Hconn, StatType, Stat, Compcode, Reason);
```

Declare the parameters as follows:

```
dc1 Hconn      fixed bin(31); /* Connection handle */
dc1 StatType   fixed bin(31); /* Status type */
dc1 Stat       like MQSTS;    /* Status information structure */
dc1 CompCode   fixed bin(31); /* Completion code */
dc1 Reason     fixed bin(31); /* Reason code qualifying CompCode */
```

System/390 Assembler invocation

```
CALL MQSTAT,(HCONN,STATTYPE,STAT,COMPCODE,REASON)
```

Declare the parameters as follows:

HCONN	DS	F	Connection handle
STATTYPE	DS	F	Status type
STAT	CMQSTSA,		Status information structure
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQSUB - Register subscription

Use the MQSUB call to register the applications subscription to a particular topic.

Syntax

MQSUB (*Hconn*, *SubDesc*, *Hobj*, *Hsub*, *Compcode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

SubDesc

Type: MQSD - input/output

This is a structure that identifies the object in use that is being registered by the application. See [“MQSD - Subscription descriptor” on page 539](#) for more information.

Hobj

Type: MQHOBJ - input/output

This handle represents the access that has been established to obtain the messages sent to this subscription. These messages can either be stored on a specific queue or the queue manager can manage their storage without using a specific queue.

To use a specific queue, you must associate it with the subscription when the subscription is created. You can do this in two ways:

- By using the DEFINE SUB MQSC command and provided that command with the name of a queue object.
- By providing this handle when calling MQSUB with the MQSO_CREATE

If this handle is provided as an input parameter on the call, it must be a valid object handle returned from a previous MQOPEN call of a queue using at least one of the following options:

- MQOO_INPUT_*
- MQOO_BROWSE
- MQOO_OUTPUT (if the queue is a remote queue)

If this is not the case, the call fails with MQRC_HOBJ_ERROR. It cannot be an object handle to an alias queue that resolves to a topic object. If so, the call fails with MQRC_HOBJ_ERROR.

If the queue manager is to manage the storage of messages sent to this subscription, this should be set when you create the subscription, by using the MQSO_MANAGED option. The queue manager then returns this handle as an output parameter on the call. The handle that is returned is known as a managed handle. If MQHO_NONE is specified but MQSO_MANAGED is not specified, the call fails with MQRC_HOBJ_ERROR.

When a managed handle is returned to you by the queue manager, you can use it on an MQGET or MQCB call with or without browse options, on an MQINQ call, or on MQCLOSE. You cannot use it on MQPUT, MQSUB, MQSET; attempting to do so fails with MQRC_NOT_OPEN_FOR_OUTPUT, MQRC_HOBJ_ERROR, or MQRC_NOT_OPEN_FOR_SET.

If this subscription is being resumed using the MQSO_RESUME option in the MQSD structure, the handle can be returned to the application in this parameter by setting MQSO_MANAGED to MQHO_NONE. You can do this whether the subscription is using a managed handle or not and it can be useful to provide subscriptions created using DEFINE SUB with the handle to the subscription queue defined on that command. In the case where an administratively created subscription is being resumed, the queue opens with MQOO_INPUT_AS_Q_DEF and MQOO_BROWSE. If you need to specify other options, the application must open the subscription queue explicitly and provide the object handle on the call. If there is a problem opening the queue the call fails with MQRC_INVALID_DESTINATION. If the *Hobj* is provided, it must be equivalent to the *Hobj* in the original MQSUB call. This means if an object handle returned from an MQOPEN call is being provided, the handle must be to the same queue as previously used. If it is not the same queue, the call fails with MQRC_HOBJ_ERROR.

If this subscription is being altered using the MQSO_ALTER option in the MQSD structure, then a different *Hobj* can be provided. Any publications that have been delivered to the queue and were previously identified through this parameter stay on that queue and it is the responsibility of the application to retrieve those messages if the *Hobj* parameter now represents a different queue.

The table summarizes the use of this parameter with various subscription options:

Options	<i>Hobj</i>	Description
MQSO_CREATE + MQSO_MANAGED	Ignored on input	Creates a subscription with storage of messages managed by the queue manager
MQSO_CREATE	A valid object handle	Creates a subscription providing a specific queue as the destination for messages.
MQSO_RESUME	MQHO_NONE	Resumes a previously created subscription whether it was managed or not, and has the queue manager return the object handle for use by the application.
MQSO_RESUME	A valid, matching, object handle	Resumes a previously created subscription that uses a specific queue as the destination for messages and use an object handle with specific open options.
MQSO_ALTER + MQSO_MANAGED	MQHO_NONE	Alters an existing subscription that was previously using a specific queue, so it is now a managed subscription. The class of destination (managed or not) cannot be changed.

Options	Hobj	Description
MQSO_ALTER	A valid object handle	Alters an existing subscription, whether it was managed or not, so that it now uses a specific queue. When the MQSO_MANAGED option is not used, the queue provided can be changed, but the class of destination (managed or not) cannot be changed.

Whether it was provided or returned, *Hobj* must be specified on subsequent MQGET or MQCB calls that want to receive the publication messages sent to this subscription.

The *Hobj* handle is no longer valid when the MQCLOSE call is issued on it, or when the unit of processing that defines the scope of the handle terminates (until the application disconnects). The scope of the object handle returned is the same as that of the connection handle specified on the call. See [Hconn \(MQHCONN\)](#) - output for information about handle scope. An MQCLOSE of the *Hobj* handle does not affect the *Hsub* handle.

Hsub

Type: MQHOBJ - output

This handle represents the subscription that has been made. It can be used for two further operations:

- It can be used on a subsequent MQSUBRQ call to request that publications be sent when the MQSO_PUBLICATIONS_ON_REQUEST option has been used when making the subscription.
- It can be used on a subsequent MQCLOSE call to remove the subscription that has been made. The *Hsub* handle ceases to be valid when the MQCLOSE call is issued, or when the unit of processing that defines the scope of the handle terminates. The scope of the object handle returned is the same as that of the connection handle specified on the call. An MQCLOSE of the *Hsub* handle does not affect the *Hobj* handle.

This handle cannot be passed to an MQGET or MQCB call. You must use the *Hobj* parameter. You cannot use this handle on any IBM MQ call other than MQCLOSE or MQSUBRQ. Passing this handle to any other IBM MQ call results in MQRC_HOBJ_ERROR.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion

MQCC_WARNING

Warning (partial completion)

MQCC_FAILED

Call failed

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK, the reason code is as follows:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED, the reason code is one of the following:

MQRC_CLUSTER_RESOLUTION_ERROR

(2189, X'88D') Cluster name resolution failed.

MQRC_DURABILITY_NOT_ALLOWED

2436 (X'0984') An MQSUB call using the MQSO_DURABLE option failed.

MQRC_FUNCTION_NOT_SUPPORTED

2298 (X'08FA') The function requested is not available in the current environment.

MQRC_HOBJ_ERROR

2019 (X'07E3') Object handle Hobj not valid.

MQRC_IDENTITY_MISMATCH

2434 (X'0982') Subscription name matches existing subscription.

MQRC_NOT_AUTHORIZED

2035 (X'07F3') The user is not authorized to perform the operation.

MQRC_NO_SUBSCRIPTION

2428 (X'097C') The identified subscription name does not exist.

MQRC_OBJECT_STRING_ERROR

2441 (X'0989') Objectstring field not valid.

MQRC_OPTIONS_ERROR

2046 (X'07FE') Options parameter or field contains options that are not valid, or a combination of options that is not valid.

MQRC_Q_MGR QUIESCING

2161 (X'0871') Queue manager quiescing.

MQRC_RECONNECT_Q_MGR_REQD

2555 (X'09FB'X) The MQCNO_RECONNECT_Q_MGR option is required.

MQRC_RETAINED_MSG_Q_ERROR

2525 (X'09DD') Retained publications which exist for the subscribed topic string, cannot be retrieved.

MQRC_RETAINED_NOT_DELIVERED

2526 (X'09DE') The retained publications which exist for the subscribed topic string, cannot be delivered to the subscription destination queue, and cannot be delivered to the dead-letter queue.

MQRC_SD_ERROR

2424 (X'0978') Subscription descriptor (MQSD) not valid.

MQRC_SELECTION_NOT_AVAILABLE

2551 (X'09F7') The selection string does not follow the IBM MQ selector syntax and no extended message selection provider was available.

MQRC_SELECTION_STRING_ERROR

2519 (X'09D7') The selection string must be specified as described in the MQCHARV structure documentation.

MQRC_SELECTOR_SYNTAX_ERROR

2459 (X'099B') An MQOPEN, MQPUT1, or MQSUB call was issued but a selection string was specified which contained a syntax error.

MQRC_SUB_USER_DATA_ERROR

2431 (X'097F') SubUserData field not valid.

MQRC_SUB_NAME_ERROR

2440 (X'0988') SubName field not valid.

MQRC_SUB_ALREADY_EXISTS

2432 (X'0980') Subscription already exists.

MQRC_SUB_USER_DATA_ERROR

2431 (X'097F') SubUserData field not valid.

MQRC_TOPIC_STRING_ERROR

2425 (X'0979') Topic string is not valid.

MQRC_UNKNOWN_OBJECT_NAME

2085 (X'0825') Object identified in the MQSD ObjectName field cannot be found.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

- The subscription is made to a topic, named either using the short name of a pre-defined topic object, the full name of the topic string, or it is formed by the concatenation of two parts. See the description of *ObjectName* and *ObjectString* in “MQSD - Subscription descriptor” on page 539.
 - The queue manager performs security checks when an MQSUB call is issued, to verify that the user identifier under which the application is running has the appropriate level of authority before access is permitted. The appropriate topic object is located in the topic hierarchy and an authority check is made on this topic object to ensure authority to subscribe is set. If the MQSO_MANAGED option is not used, an authority check is made on the destination queue to ensure that authority for output is set. If the MQSO_MANAGED option is used, no authority check is made on the managed queue for output or inquire access.
 - If you do not provide an Hobj as input, the MQSUB call allocates two handles, an object handle (Hobj) and a subscription handle (Hsub).
 - The Hobj returned on the MQSUB call when the MQSO_MANAGED option is used, can be inquired in order to find out attributes such as the Backout threshold and the Excessive backout requeue name. You can also inquire the name of the managed queue, but you must not attempt to directly open this queue.
 - Subscriptions can be grouped allowing only a single publication to be delivered to the group of subscriptions even where more than one of the group matched the publication. Subscriptions are grouped using the MQSO_GROUP_SUB option and in order to group subscriptions they must be
 - using the same named queue (that is not using the MQSO_MANAGED option) on the same queue manager - represented by the Hobj parameter on the MQSUB call
 - share the same SubCorrelId
 - be of the same SubLevel
- These attributes define the set of subscriptions considered to be in the group, and are also the attributes that cannot be altered if a subscription is grouped. Alteration of SubLevel results in MQRC_SUBLEVEL_NOT_ALTERABLE, and alteration of any of the others (which can be changed if a subscription is not grouped) results in MQRC_GROUPING_NOT_ALTERABLE.
- Successful completion of the MQSUB call does not mean that the action completed. To check that this call has completed, see the [DEFINE SUB](#) step in [Checking that async commands for distributed networks have finished](#).
 - Fields in the MQSD are filled in on return from an MQSUB call which uses the MQSO_RESUME option. The MQSD returned can be passed directly into an MQSUB call which uses the MQSO_ALTER option with any changes you need to make to the subscription applied to the MQSD. Some fields have special considerations as noted in the table.

MQSD output from MQSUB	
Field name in MQSD	Special considerations
Access or creation options	Some of the options can be reset on return from the MQSUB call. If you then reuse the MQSD in an MQSUB call, the option you require must be explicitly set.
Durability options, Destination options, Registration Options & Wildcard options	These options are set as appropriate
Publication options	These options are set as appropriate, except for MQSO_NEW_PUBLICATIONS_ONLY which is only applicable to MQSO_CREATE.

MQSD output from MQSUB (<i>continued</i>)	
Field name in MQSD	Special considerations
Other options	These options are unchanged on return from an MQSUB call. They control how the API call is issued and are not stored with the subscription. They must be set as required on any subsequent MQSUB call reusing the MQSD.
ObjectName	This input only field is unchanged on return from an MQSUB call.
ObjectString	This input only field is unchanged on return from an MQSUB call. The Full topic name used is returned in the <i>ResObjectString</i> field, if a buffer is provided.
AlternateUserId and AlternateSecurityId	These input only fields are unchanged on return from an MQSUB call. They control how the API call is issued and are not stored with the subscription. They must be set as required on any subsequent MQSUB call reusing the MQSD.
SubExpiry	On return from an MQSUB call using the MQSO_RESUME option, this field is set to the original expiry of the subscription and not the remaining expiry time. If you then reuse the MQSD in an MQSUB call using the MQSO_ALTER option you reset the expiry of the subscription to start counting down again.
SubName	This field is an input field on an MQSUB call and is not changed on output.
SubUserData and SelectionString	<p>These variable length fields are returned on output from an MQSUB call using the MQSO_RESUME option, if a buffer is provided, and also a positive buffer length in <i>VSBufSize</i>. If no buffer is provided only the length is returned in the <i>VSLength</i> field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only <i>VSBufSize</i> bytes are returned in the provided buffer.</p> <p>If you then reuse the MQSD in an MQSUB call using the MQSO_ALTER option and a buffer is not provided but a non-zero <i>VSLength</i> is provided, if that length matches the existing length of the field, no alteration is made to the field.</p>
SubCorrelId and PubAccountingToken	<p>If you do not use MQSO_SET_CORREL_ID, then the <i>SubCorrelId</i> is generated by the queue manager. If you do not use MQSO_SET_IDENTITY_CONTEXT, then the <i>PubAccountingToken</i> is generated by the queue manager.</p> <p>These fields are returned in the MQSD from an MQSUB call using the MQSO_RESUME option. If they are generated by the queue manager, the generated value is returned on an MQSUB call using the MQSO_CREATE or MQSO_ALTER option.</p>
PubPriority, SubLevel & PubApplIdentityData	These fields are returned in the MQSD.

MQSD output from MQSUB (<i>continued</i>)	
Field name in MQSD	Special considerations
ResObjectString	This output only field is returned in the MQSD if a buffer is provided.

C invocation

```
MQSUB (Hconn, &SubDesc, &Hobj, &Hsub, &CompCode, &Reason)
```

Declare the parameters as follows:

```
MQHCONN Hconn;    /* Connection handle */
MQSD     SubDesc;  /* Subscription descriptor */
MQHOBJ   Hobj;     /* Object handle */
MQHOBJ   Hsub;     /* Subscription handle */
MQLONG   CompCode; /* Completion code */
MQLONG   Reason;   /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQSUB' USING HCONN, SUBDESC, HOBJ, HSUB, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN    PIC S9(9) BINARY.
** Subscription descriptor
01 SUBDESC.
   COPY CMQSDV.
** Object handle
01 HOBJ     PIC S9(9) BINARY.
** Subscription handle
01 HSUB     PIC S9(9) BINARY.
** Completion code
01 COMPCODE PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON   PIC S9(9) BINARY.
```

PL/I invocation

```
call MQSUB (Hconn, SubDesc, Hobj, Hsub, CompCode, Reason)
```

Declare the parameters as follows:

```
dcl Hconn    fixed bin(31); /* Connection handle */
dcl SubDesc  like MQSD;    /* Subscription descriptor */
dcl Hobj     fixed bin(31); /* Object handle */
dcl Hsub     fixed bin(31); /* Subscription handle */
dcl CompCode fixed bin(31); /* Completion code */
dcl Reason   fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQSUB,(HCONN,SUBDESC,HOBJ,HSUB,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN    DS      F  Connection handle
SUBDESC  CMQSDA  ,  Subscription descriptor
HOBJ     DS      F  Object handle
HSUB     DS      F  Subscription handle
COMPCODE DS      F  Completion code
REASON   DS      F  Reason code qualifying COMPCODE
```

MQSUBRQ - Subscription request

Use the MQSUBRQ call to make a request for the retained publication, when the subscriber has been registered with MQSO_PUBLICATIONS_ON_REQUEST.

Syntax

MQSUBRQ (*Hconn*, *Hsub*, *Action*, *SubRqOpts*, *Compcode*, *Reason*)

Parameters

Hconn


Type: MQHCONN - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *Hconn* :

MQHC_DEF_HCONN

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Hsub

Type: MQHOBJ - input

This handle represents the subscription for which an update is to be requested. The value of *Hsub* was returned from a previous MQSUB call.

Action

Type: MQLONG - input

This parameter controls the particular action that is being requested on the subscription. The following value must be specified:

MQSR_ACTION_PUBLICATION

This action requests that an update publication is sent for the specified topic. It can be used only if the subscriber specified the option MQSO_PUBLICATIONS_ON_REQUEST on the MQSUB call when it made the subscription. If the queue manager has a retained publication for the topic, this is sent to the subscriber. If not, the call fails. If an application is sent a publication which was retained, this is indicated by the MQIsRetained message property of that publication.

Since the topic in the existing subscription represented by the *Hsub* parameter can contain wildcards, the subscriber might receive multiple retained publications.

SubRqOpts

Type: MQSRO - input/output

These options control the action of MQSUBRQ, see [“MQSRO - Subscription request options” on page 562](#) for details.

If no options are required, programs written in C or S/390 assembler can specify a null parameter address instead of specifying the address of an MQSRO structure.

CompCode

Type: MQLONG - output

The completion code; it is one of the following:

MQCC_OK

Successful completion

MQCC_WARNING

Warning (partial completion)

MQCC_FAILED

Call failed

Reason

Type: MQLONG - output

The reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_FUNCTION_NOT_SUPPORTED

2298 (X'08FA') The function requested is not available in the current environment.

MQRC_NO_RETAINED_MSG

2437 (X'0985') There are no retained publications currently stored for this topic.

MQRC_OPTIONS_ERROR

2046 (X'07FE') Options parameter or field contains options that are not valid, or a combination of options that is not valid.

MQRC_Q_MGR QUIESCING

2161 (X'0871') Queue manager quiescing.

MQRC_SRO_ERROR

2438 (X'0986') On the MQSUBRQ call, the Subscription Request Options MQSRO is not valid.

MQRC_RETAINED_MSG_Q_ERROR

2525 (X'09DD') Retained publications which exist for the subscribed topic string, cannot be retrieved.

MQRC_RETAINED_NOT_DELIVERED

2526 (X'09DE') The retained publications which exist for the subscribed topic string, cannot be delivered to the subscription destination queue, and cannot be delivered to the dead-letter queue.

For detailed information about these codes, see [Messages and reason codes](#).

Usage notes

The following usage notes apply to the use of the Action code MQSR_ACTION_PUBLICATION:

1. If this verb completes successfully, the retained publications matching the subscription specified have been sent to the subscription and can be received by using MQGET or MQCB using the Hobj returned on the original MQSUB verb that created the subscription.
2. If the topic subscribed to by the original MQSUB verb that created the subscription contained a wildcard, more than one retained publication can be sent. The number of publications sent as a result of this call is recorded in the NumPubs field in the SubRqOpts structure.
3. If this verb completes with a reason code of MQRC_NO_RETAINED_MSG then there were no currently retained publications for the topic specified.#
4. If this verb completes with a reason code of MQRC_RETAINED_MSG_Q_ERROR or MQRC_RETAINED_NOT_DELIVERED then there are currently retained publications for the topic specified but an error has occurred that that meant they were unable to be delivered.
5. The application must have a current subscription to the topic before it can make this call. If the subscription was made in a previous instance of the application and a valid handle to the subscription is not available, the application must first call MQSUB with the MQSO_RESUME option to obtain a handle to it for use in this call.

6. The publications are sent to the destination that is registered for use with the current subscription of this application. If the publications must be sent somewhere else, the subscription must first be altered using the MQSUB call with the MQSO_ALTER option.

C invocation

```
MQSUB (Hconn, Hsub, Action, &SubRqOpts, &CompCode, &Reason)
```

Declare the parameters as follows:

```
MQHCONN Hconn;      /* Connection handle */
MQHOBJ  Hsub;        /* Subscription handle */
MQLONG  Action;      /* Action requested by MQSUBRQ */
MQSRO   SubRqOpts;   /* Options that control the action of MQSUBRQ */
MQLONG  CompCode;    /* Completion code */
MQLONG  Reason;      /* Reason code qualifying CompCode */
```

COBOL invocation

```
CALL 'MQSUBRQ' USING HCONN, HSUB, ACTION, SUBRQOPTS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle
01 HCONN PIC S9(9) BINARY.
** Subscription handle
01 HSUB PIC S9(9) BINARY.
** Action requested by MQSUBRQ
01 ACTION PIC S9(9) BINARY.
** Options that control the action of MQSUBRQ
01 SUBRQOPTS.
COPY CMQSROV.
** Completion code
01 COMPCODE PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
01 REASON PIC S9(9) BINARY.
```

PL/I invocation

```
call MQSUBRQ (Hconn, Hsub, Action, SubRqOpts, CompCode, Reason)
```

Declare the parameters as follows:

```
dcl Hconn fixed bin(31); /* Connection handle */
dcl Hsub fixed bin(31);  /* Subscription handle */
dcl Action fixed bin(31); /* Action requested by MQSUBRQ */
dcl SubRqOpts like MQSRO; /* Options that control the action of MQSUBRQ */
dcl CompCode fixed bin(31); /* Completion code */
dcl Reason fixed bin(31); /* Reason code qualifying CompCode */
```

High Level Assembler invocation

```
CALL MQSUBRQ,(HCONN, HSUB, ACTION, SUBRQOPTS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN DS F Connection handle
HSUB DS F Subscription handle
ACTION DS F Action requested by MQSUBRQ
SUBRQOPTS CMQSROA , Options that control the action of MQSUBRQ
COMPCODE DS F Completion code
REASON DS F Reason code qualifying COMPCODE
```

Attributes of objects

This collection of topics lists only those IBM MQ objects that can be the subject of an MQINQ function call, and gives details of the attributes that can be inquired on and the selectors to be used.

Attributes for the queue manager

Some queue-manager attributes are fixed for particular implementations; others can be changed by using the MQSC command ALTER QMGR.

The attributes can also be displayed by using the command DISPLAY QMGR. Most queue-manager attributes can be inquired by opening a special MQOT_Q_MGR object, and using the MQINQ call with the handle returned.

The following table summarizes the attributes that are specific to the queue manager. The attributes are described in alphabetical order.

Note: The names of the attributes shown in this section are descriptive names used with the MQINQ call; the names are the same as for the PCF commands. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) Commands](#) for more information.

Table 560. Attributes for the queue manager	
Attribute	Description
AccountingConnOverride	Override accounting settings.
AccountingInterval	How often to write intermediate accounting records.
ActivityConnOverride	Override activity settings.
ActivityTrace	Controls the collection of IBM MQ MQI application activity trace.
AdoptNewMCACheck	Elements checked to determine whether to adopt new MCA.
AdoptNewMCAType	Whether to restart automatically an orphaned instance of an MCA of a particular channel type.
AlterationDate	Date when definition was last changed
AlterationTime	Time when definition was last changed
AuthorityEvent	Controls whether authorization (Not Authorized) events are generated
BridgeEvent	Control attribute for bridge events.
ChannelAutoDef	Controls whether automatic channel definition is permitted
ChannelAutoDefEvent	Controls whether channel automatic-definition events are generated
ChannelAutoDefExit	Name of user exit for automatic channel definition
ChannelEvent	Control attribute for channel events.
ChannelInitiatorControl	Control attribute for channel initiator
ChannelMonitoring	Online monitoring data for channels
ChannelStatistics	Controls collection of statistics data for channels.
ChinitAdapters	Number of adapter subtasks for processing IBM MQ calls.
ChinitDispatchers	Number of dispatchers to use for the channel initiator.
	Reserved for IBM use.
ChinitTraceAutoStart	Whether channel initiator trace should start automatically.
ChinitTraceTableSize	Size of channel initiator's trace data space.
ClusterSenderMonitoringDefault	Online monitoring data default for cluster sender channels
ClusterSenderStatistics	Controls collection of statistics monitoring information for cluster sender channels.
ClusterWorkloadData	User data for cluster workload exit
ClusterWorkloadExit	Name of user exit for cluster workload management
ClusterWorkloadLength	Maximum length of message data passed to cluster workload exit
CLWLMRUChannels	Number of most recently used channels for cluster workload balancing

Table 560. Attributes for the queue manager (continued)

Attribute	Description
CLWLUseQ	Cluster workload use remote queue.
CodedCharSetId	Coded character set identifier
CommandEvent	Control attribute for command events.
CommandInputQName attribute	Command input queue name
CommandLevel	Command level
CommandServerControl attribute	Control attribute for command server.
Configuration Event attribute	Control attribute for configuration events.
DeadLetterQName	Name of dead-letter queue
DEFCLXQ	Default cluster transmission queue type
DefXmitQName	Default transmission queue name
DistLists	Distribution list support
DNSGroup	Name of group for TCP listener when using Workload Manager Dynamic Domain Name Services support.
DNSWLM	Whether TCP listener registers with Workload Manager for Dynamic Domain Name Services.
ExpiryInterval	Interval between scans for expired messages
IGQPutAuthority	Intra-group queuing put authority
IGQUserId	Intra-group queuing user identifier
InhibitEvent	Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated
IPAddressVersion	Version of the Internet Protocol address
IntraGroupqueuing	Intra-group queuing support
ListenerTimer	Time interval between attempts to restart listener after APPC or TCP/IP failure.
LocalEvent	Controls whether local error events are generated
LoggerEvent	Controls whether logger events are generated
LUGroupName	Generic LU name for LU 6.2 listener that handles inbound transmissions for queue-sharing group.
LUName	Name of LU to use for outbound LU 6.2 transmissions.
LU62ARMSuffix	Suffix of SYS1.PARMLIB member APPCPMxx, that nominates LUADD for this channel initiator.
LU62Channels	Maximum number of current channels or connected clients that use LU 6.2.
MaxActiveChannels	Maximum number of channels that can be active at any time.
MaxChannels	Maximum number of current channels.
MaxHandles	Maximum number of handles
MaxMsgLength	Maximum message length in bytes
MaxPriority attribute	Maximum priority
MaxPropertiesLength	Maximum length of property data in bytes
MaxUncommittedMsgs	Maximum number of uncommitted messages within a unit of work
MQIAccounting	Controls collection of accounting information for MQI data.
MQIStatistics	Controls collection of statistics monitoring information for queue manager.
MsgMarkBrowseInterval	Interval after which the queue manager can remove the mark from browsed messages.
OutboundPortMin	With <i>OutboundPortMin</i> , defines range of port numbers to use when binding outgoing channels.
OutboundPortMin	With <i>OutboundPortMax</i> , defines range of port numbers to use when binding outgoing channels.
PerformanceEvent	Controls whether performance-related events are generated
Platform	Platform on which the queue manager is running
PubSubNPInputMsg	Whether to discard (or keep) an undelivered input message
PubSubNPResponse	Controls the behavior of undelivered
PubSubMaxMsgRetryCount	The number of retries when processing (under syncpoint) a failed command message

Table 560. Attributes for the queue manager (continued)

Attribute	Description
PubSubSyncPoint	Whether only persistent (or all) messages should be processed under syncpoint
PubSubMode	Whether the queued publish/subscribe interface is running
QMGrDesc	Queue manager description
QMGrIdentifier	Unique internally generated identifier of queue manager
QMGrName	Queue manager name
QSGName	Name of queue-sharing group
QueueAccounting	Controls collection of accounting information for queues.
QueueMonitoring	Online monitoring data for queues
QueueStatistics	Controls collection of statistics data for queues.
ReceiveTimeout	How long TCP/IP channel waits for data before returning to inactive state.
ReceiveTimeoutMin	Qualifier for <i>ReceiveTimeout</i> .
ReceiveTimeoutType	Minimum time that TCP/IP channel waits for data before returning to inactive state.
RemoteEvent	Controls whether remote error events are generated
RepositoryName	Name of cluster for which this queue manager provides repository services
RepositoryNamelist	Name of namelist object containing names of clusters for which this queue manager provides repository services
ScyCase	Case of security profiles
SharedQMGrName	Shared queue queue-manager name
“SPLCAP” on page 797	IBM MQ Advanced Message security protection for a queue manager turned on or off.
SSLCRLNamelist 1	Name of namelist object containing names of authentication information objects.
SSLCryptoHardware 1	Cryptographic hardware configuration string.
SSEvent	Control attribute for SSL events.
SSLFIPSRequired	Use only FIPS-certified algorithms for cryptography.
SSLKeyRepository 1	Location of SSL key repository.
SSLKeyResetCount	SSL key reset count.
SSLTasks 1	Number of server subtasks for processing SSL calls.
StatisticsInterval	How often to write statistics monitoring data.
StartStopEvent	Controls whether start and stop events are generated
SyncPoint	Syncpoint availability
TCPChannels	Maximum number of current channels or connected clients that use TCP/IP.
TCPKeepAlive	Whether to use TCP KEEPALIVE to check other end of connection.
TCPName	Name of TCP/IP system that you are using.
TCPStackType	How channel initiator can use TCP/IP addresses.
TraceRouteRecording attribute	Controls recording of trace-route information.
TriggerInterval	Trigger-message interval
Version	Version
XrCapability	Specifies whether Telemetry commands are supported.
Notes:	
1. This attribute cannot be inquired using the MQINQ call, and is not described in this section. See Change Queue Manager for details of this attribute.	

Related tasks

[Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client](#)

Related reference

[Federal Information Processing Standards \(FIPS\) for UNIX, Linux and Windows](#)

AccountingConnOverride (MQLONG)

This allows applications to override the setting of the ACCTMQI and ACCTQDATA values in the Qmgr attribute.

The value is one of the following:

MQMON_DISABLED

Applications cannot override the setting of the ACCTMQI and ACCTQ Qmgr attributes using the Options field in the MQCNO structure on the MQCONN call. This is the default value.

MQMON_ENABLED

Applications can override the ACCTQ and ACCTMQI Qmgr attributes using the Options field in the MQCNO structure.

Changes to this value are only effective for connections to the queue manager after the change to the attribute.

This attribute is supported only on IBM i, Unix systems, and Windows.

To determine the value of this attribute, use the MQIA_ACCOUNTING_CONN_OVERRIDE selector with the MQINQ call.

AccountingInterval (MQLONG)

This specifies how long before intermediate accounting records are written (in seconds).

The value is an integer in the range 0 to 604800, with a default value of 1800 (30 minutes). Specify 0 to turn off intermediate records.

This attribute is supported only on IBM i, Windows, UNIX, and Linux systems.

To determine the value of this attribute, use the MQIA_ACCOUNTING_INTERVAL selector with the MQINQ call.

ActivityConnOverride (MQLONG)

This allows applications to override the setting of the ACTVTRC value in the queue manager attribute.

The value is one of the following:

MQMON_DISABLED

Applications cannot override the setting of the ACTVTRC queue manager attribute using the Options field in the MQCNO structure on the MQCONN call. This is the default value.

MQMON_ENABLED

Applications can override the ACTVTRC queue manager attribute using the Options field in the MQCNO structure.

Changes to this value are only effective for connections to the queue manager after the change to the attribute.

This attribute is supported only on IBM i, Unix systems, and Windows.

To determine the value of this attribute, use the MQIA_ACTIVITY_CONN_OVERRIDE selector with the MQINQ call.

ActivityTrace (MQLONG)

This controls the collection of IBM MQ MQI application activity trace.

The value is one of the following:

MQMON_ON

Collect IBM MQ MQI application activity trace.

MQMON_OFF

Do not collect IBM MQ MQI application activity trace. This is the default value.

If you set the queue manager attribute ACTVCON0 to ENABLED, this value might be overridden for individual connections using the Options field in the MQCNO structure.

Changes to this value are only effective for connections to the queue manager after the change to the attribute.

This attribute is supported only on IBM i, Unix systems, and Windows.

To determine the value of this attribute, use the MQIA_ACTIVITY_TRACE selector with the MQINQ call.

AdoptNewMCACheck (MQLONG)

This defines the elements to check to determine whether to adopt an MCA when a new inbound channel is detected that has the same name as an MCA that is already active

The value is one of the following:

MQADOPT_CHECK_Q_MGR_NAME

Check the queue manager name.

MQADOPT_CHECK_NET_ADDR

Check the network address.

MQADOPT_CHECK_ALL

Check the queue manager name and network address. If possible, perform this check to protect your channels from being shut down, inadvertently or maliciously. This is the default value.

MQADOPT_CHECK_NONE

Do not check any elements.

Changes to this attribute take effect the next time that a channel attempts to adopt a channel.

This attribute is supported only on z/OS.

To determine the value of this attribute, use the MQIA_ADOPTNEWMCA_CHECK selector with the MQINQ call.

AdoptNewMCAType (MQLONG)

This specifies whether to restart automatically an orphaned instance of an MCA of a particular channel type when a new inbound channel request matching the AdoptNewMCACheck attribute is detected

It is one of the following values:

MQADOPT_TYPE_NO

Adopting orphaned channel instances is not required. This is the default value.

MQADOPT_TYPE_ALL

Adopt all channel types.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_ADOPTNEWMCA_TYPE selector with the MQINQ call.

AlterationDate (MQCHAR12)

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the MQCA_ALTERATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_DATE_LENGTH.

AlterationTime (MQCHAR8)

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the MQCA_ALTERATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_TIME_LENGTH.

AuthorityEvent (MQLONG)

This controls whether authorization (Not Authorized) events are generated. It is one of the following values:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_AUTHORITY_EVENT selector with the MQINQ call.

BridgeEvent (MQLONG)

This specifies whether IMS bridge events are generated.

The value is one of the following:

MQEVR_ENABLED

Generate IMS bridge events, as follows:

MQRC_BRIDGE_STARTED

MQRC_BRIDGE_STOPPED

MQEVR_DISABLED

Do not generate IMS bridge events; this is the default value.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_BRIDGE_EVENT selector with the MQINQ call.

ChannelAutoDef (MQLONG)

This attribute controls the automatic definition of channels of type MQCHT_RECEIVER and MQCHT_SVRCONN. Automatic definition of MQCHT_CLUSSDR channels is always enabled. The value is one of the following:

MQCHAD_DISABLED

Channel auto-definition disabled.

MQCHAD_ENABLED

Channel auto-definition enabled.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, and Windows.

To determine the value of this attribute, use the MQIA_CHANNEL_AUTO_DEF selector with the MQINQ call.

ChannelAutoDefEvent (MQLONG)

This controls whether channel automatic-definition events are generated. It applies to channels of type MQCHT_RECEIVER, MQCHT_SVRCONN, and MQCHT_CLUSSDR. The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, and Windows.

To determine the value of this attribute, use the MQIA_CHANNEL_AUTO_DEF_EVENT selector with the MQINQ call.

ChannelAutoDefExit (MQCHARn)

This is the name of the user exit for automatic channel definition. If this name is nonblank, and *ChannelAutoDef* has the value MQCHAD_ENABLED, the exit is called each time that the queue manager is about to create a channel definition. This applies to channels of type MQCHT_RECEIVER, MQCHT_SVRCONN, and MQCHT_CLUSSDR. The exit can then do one of the following:

- Create the channel definition without change.
- Modify the attributes of the channel definition that is created.
- Suppress creation of the channel entirely.

Note: Both the length and the value of this attribute are environment specific. See the introduction to the MQCD structure in [“MQCD - Channel definition” on page 1487](#) for details of the value of this attribute in various environments.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows, and z/OS. On z/OS, it applies only to cluster-sender and cluster-receiver channels.

To determine the value of this attribute, use the MQCA_CHANNEL_AUTO_DEF_EXIT selector with the MQINQ call. The length of this attribute is given by MQ_EXIT_NAME_LENGTH.

ChannelEvent (MQLONG)

This specifies whether channel events are generated.

It is one of the following values:

MQEVR_EXCEPTION

Only generate the following channel events:

- MQRC_CHANNEL_ACTIVATED
- MQRC_CHANNEL_CONV_ERROR
- MQRC_CHANNEL_NOT_ACTIVATED
- MQRC_CHANNEL_STOPPED with the following ReasonQualifiers:

MQRQ_CHANNEL_STOPPED_ERROR
MQRQ_CHANNEL_STOPPED_RETRY
MQRQ_CHANNEL_STOPPED_DISABLED

MQRC_CHANNEL_STOPPED_BY_USER

MQEVR_ENABLED

Generate all channel events. That is, in addition to those generated by EXCEPTION, generate the following channel events:

- MQRC_CHANNEL_STARTED
- MQRC_CHANNEL_STOPPED with the following ReasonQualifier:

MQRQ_CHANNEL_STOPPED_OK

MQEVR_DISABLED

Do not generate channel events; this is the default value.

To determine the value of this attribute, use the MQIA_CHANNEL_EVENT selector with the MQINQ call.

ChannelInitiatorControl (MQLONG)

This specifies whether the channel initiator is to be started when the queue manager starts.

It is one of the following values:

MQSVC_CONTROL_MANUAL

The channel initiator is not to be started automatically.

MQSVC_CONTROL_Q_MGR

The channel initiator is to be started automatically when the queue manager starts.

To determine the value of this attribute, use the MQIA_CHINIT_CONTROL selector with the MQINQ call.

ChannelMonitoring (MQLONG)

This attribute specifies online monitoring data for channels.

The value is one of the following:

MQMON_NONE

Disable data collection for channel monitoring for all channels regardless of the setting of the MONCHL channel attribute. This is the default value.

MQMON_OFF

Turn monitoring data collection off for channels that specify QMGR in the MONCHL channel attribute.

MQMON_LOW

Turn monitoring data collection on with a low ratio of data collection for channels specifying QMGR in the MONCHL channel attribute.

MQMON_MEDIUM

Turn monitoring data collection on with a moderate ratio of data collection for channels specifying QMGR in the MONCHL channel attribute.

MQMON_HIGH

Turn monitoring data collection on with a high ratio of data collection for channels specifying QMGR in the MONCHL channel attribute.

On z/OS, this parameter simply turns on statistics data collection, regardless of the value you select. Specifying LOW, MEDIUM, or HIGH makes no difference to your results.

To determine the value of this attribute, use the MQIA_MONITORING_CHANNEL selector with the MQINQ call.

ChannelStatistics (MQLONG)

This controls the collection of statistics data for channels.

The value is one of the following:

MQMON_NONE

Disable data collection for channel statistics for all channels regardless of the setting of the STATCHL channel attribute. This is the default value.

MQMON_OFF

Turn statistics data collection off for channels that specify QMGR in the STATCHL channel attribute.

MQMON_LOW

Turn statistics data collection on with a low ratio of data collection for channels specifying QMGR in the STATCHL channel attribute.

MQMON_MEDIUM

Turn statistics data collection on with a moderate ratio of data collection for channels specifying QMGR in the STATCHL channel attribute.

MQMON_HIGH

Turn statistics data collection on with a high ratio of data collection for channels specifying QMGR in the STATCHL channel attribute.

For most systems you are recommended to use MEDIUM. However, for a channel that processes a high volume of messages each second, you might want to reduce the sampling level by selecting LOW. Also, for a channel that processes only a few messages, and for which the most current information is important, you might want to select HIGH.

On z/OS, this parameter simply turns on statistics data collection, regardless of the value you select. Specifying LOW, MEDIUM, or HIGH makes no difference to your results.

To determine the value of this attribute, use the MQIA_STATISTICS_CHANNEL selector with the MQINQ call.

ChinitAdapters (MQLONG)

This is the number of adapter subtasks to use to process IBM MQ calls. The value must be 0 - 9999, with a default value of 8.

The ratio of adapters to dispatchers (the ChinitDispatchers attribute) should be about 8 to 5. However, if you have only few channels, you do not have to decrease the value of this parameter from the default value. You can use the following values: for a test system, 8 (default); for a production system, 20. Ideally, you should have 20 adapters, which gives greater parallelism of IBM MQ calls. This is important for persistent messages. Fewer adapters might be better for nonpersistent messages.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_CHINIT_ADAPTERS selector with the MQINQ call.

ChinitDispatchers (MQLONG)

This is the number of dispatchers to use for the channel initiator. The value must be 0 - 9999, with a default value of 5.

As a guideline, allow one dispatcher for 50 current channels. However, if you have only few channels, you do not have to decrease the value of this attribute from the default value. If you are using TCP/IP, the greatest number of dispatchers that are used for TCP/IP channels is 100, even if you specify a larger value here. You can use the following settings: test systems, 5 (the default); production systems, 20 (you need 20 dispatchers to handle up to 1000 active channels).

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_CHINIT_DISPATCHERS selector with the MQINQ call.

ChinitTraceAutoStart (MQLONG)

This specifies whether to start channel initiator trace automatically.

The value is one of the following:

MQTRAXSTR_YES

Start channel initiator trace automatically. This is the default value.

MQTRAXSTR_NO

Do not start channel initiator trace automatically.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_CHINIT_TRACE_AUTO_START selector with the MQINQ call.

ChinitTraceTableSize (MQLONG)

This is the size of the channel initiator's trace data space (in MB).

The value must be in the range 0 through 2048, with a default value of 2.

Note: Whenever you use large z/OS data spaces, ensure that you have sufficient auxiliary storage on your system to support any related z/OS paging activity. You might also need to increase the size of your SYS1.DUMP data sets.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_CHINIT_TRACE_TABLE_SIZE selector with the MQINQ call.

ClusterSenderMonitoringDefault (MQLONG)

This specifies the value to be substituted for the *ChannelMonitoring* attribute of automatically-defined cluster sender channels.

The value is one of the following:

MQMON_Q_MGR

Collection of online monitoring data is inherited from the setting of the queue manager *ChannelMonitoring* attribute. This is the default value.

MQMON_OFF

Monitoring for the channel is switched off

MQMON_LOW

Unless *ChannelMonitoring* is MQMON_NONE, monitoring is switched on with a low rate of data collection with a minimal effect on system performance. The data collected is not likely to be the most current.

MQMON_MEDIUM

Unless *ChannelMonitoring* is MQMON_NONE, monitoring is switched on with a moderate rate of data collection with limited effect on system performance.

MQMON_HIGH

Unless *ChannelMonitoring* is MQMON_NONE, monitoring is switched on with a high rate of data collection with a likely effect on system performance. The data collected is the most current available.

To determine the value of this attribute, use the MQIA_MONITORING_AUTO_CLUSSDR selector with the MQINQ call.

ClusterSenderStatistics (MQLONG)

Because cluster sender channels can be automatically defined from the definition of CLUSRCVR in the repository, you cannot alter the setting of the STATCHL attribute for these auto-defined cluster sender channels using ALTER channel. For these channels the decision of whether to collect online monitoring data is based on the setting of this queue manager attribute.

The value is one of the following:

MQMON_Q_MGR

Statistics data collection for auto-defined cluster sender channels is based on the value of the queue manager attribute STATCHL. This is the default value.

MQMON_OFF

Switch off statistics data collection for auto-defined cluster sender channels.

MQMON_LOW

Switch on statistics data collection for auto-defined cluster sender channels with a low ratio of data collection.

MQMON_MEDIUM

Switch on statistics data collection for auto-defined cluster sender channels with a moderate ratio of data collection.

MQMON_HIGH

Switch on statistics data collection for auto-defined cluster sender channels with a high ratio of data collection.

For most systems we recommend MEDIUM. However, for an auto-defined cluster sender channel that processes a high volume of messages each second, you might want to reduce the sampling level by selecting LOW. Also, for a channel that processes only a few messages, and for which the most current information is important, you might want to select HIGH.

On z/OS, this parameter simply turns on statistics data collection, regardless of the value you select. Specifying LOW, MEDIUM, or HIGH makes no difference to your results.

To determine the value of this attribute, use the MQIA_STATISTICS_AUTO_CLUSSDR selector with the MQINQ call.

ClusterWorkloadData (MQCHAR32)

This is a user-defined 32-byte character string that is passed to the cluster workload exit when it is called. If there is no data to pass to the exit, the string is blank.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows and z/OS.

To determine the value of this attribute, use the MQCA_CLUSTER_WORKLOAD_DATA selector with the MQINQ call.

ClusterWorkloadExit (MQCHARn)

This is the name of the user exit for cluster workload management. If this name is not blank, the exit is called each time that a message is put to a cluster queue or moved from one cluster-sender queue to another. The exit can then either accept the queue instance selected by the queue manager as the destination for the message, or select another queue instance.

Note: Both the length and the value of this attribute are environment specific.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows and z/OS.

To determine the value of this attribute, use the MQCA_CLUSTER_WORKLOAD_EXIT selector with the MQINQ call. The length of this attribute is given by MQ_EXIT_NAME_LENGTH.

ClusterWorkloadLength (MQLONG)

This is the maximum length of message data that is passed to the cluster workload exit. The actual length of data passed to the exit is the minimum of the following:

- The length of the message.
- The queue-manager's *MaxMsgLength* attribute.
- The *ClusterWorkloadLength* attribute.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows and z/OS.

To determine the value of this attribute, use the MQIA_CLUSTER_WORKLOAD_LENGTH selector with the MQINQ call.

CLWLMRUChannels (MQLONG)

This specifies the maximum number of most-recently-used cluster channels, to be considered for use by the cluster workload choice algorithm.

This is a value in the range 1 through 999999999.

To determine the value of this attribute, use the MQIA_CLWL_MRU_CHANNELS selector with the MQINQ call.

CLWLUseQ (MQLONG)

This specifies whether to use remote queues for the cluster workload.

The value is one of the following:

MQCLWL_USEQ_ANY

Use both local and remote queues.

MQCLWL_USEQ_LOCAL

Do not use remote queues. This is the default value.

To determine the value of this attribute, use the MQIA_CLWL_USEQ selector with the MQINQ call.

CodedCharSetId (MQLONG)

This defines the character set used by the queue manager for all character string fields defined in the MQI such as the names of objects, and queue creation date and time. The character set must be one that has

single-byte characters for the characters that are valid in object names. It does not apply to application data carried in the message. The value depends on the environment:

- On z/OS, the value is set from the system parameters when the queue manager is started; the default value is 500.
- On Windows, the value is the primary CODEPAGE of the user creating the queue manager.
- On IBM i, the value is that which is set in the environment when the queue manager is first created.
- On UNIX systems, the value is the default CODESET for the locale of the user creating the queue manager.

To determine the value of this attribute, use the MQIA_CODED_CHAR_SET_ID selector with the MQINQ call.

CommandEvent (MQLONG)

This specifies whether command events are generated, as follows:

MQEVR_DISABLED

Do not generate command events. This is the default.

MQEVR_ENABLED

Generate command events.

MQEVR_NO_DISPLAY

Command events are generated for all successful commands other than MQINQ.

To determine the value of this attribute, use the MQIA_COMMAND_EVENT selector with the MQINQ call.

CommandInputQName (MQCHAR48)

This is the name of the command input queue defined on the local queue manager. This is a queue to which users can send commands, if authorized to do so. The name of the queue depends on the environment:

- On z/OS, the name of the queue is SYSTEM.COMMAND.INPUT; MQSC and PCF commands can be sent to it. See [The MQSC commands](#) for details of MQSC commands and [Definitions of the Programmable Command Formats](#) for details of PCF commands.
- In all other environments, the name of the queue is SYSTEM.ADMIN.COMMAND.QUEUE, and only PCF commands can be sent to it. However, an MQSC command can be sent to this queue if the MQSC command is enclosed within a PCF command of type MQCMD_ESCAPE. See [Escape](#) for information about the Escape command.

To determine the value of this attribute, use the MQCA_COMMAND_INPUT_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

CommandLevel (MQLONG)

This indicates the level of system control commands supported by the queue manager. This can be one of the following values:

MQCMDL_LEVEL_600

Level 600 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 6.0
- IBM WebSphere MQ for HP-UX Version 6.0
- IBM WebSphere MQ for iSeries Version 6.0
- IBM WebSphere MQ for Linux Version 6.0
- IBM WebSphere MQ for Solaris Version 6.0

- IBM WebSphere MQ for Windows Version 6.0
- IBM WebSphere MQ for z/OS Version 6.0

MQCMDL_LEVEL_700

Level 700 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 7.0
- IBM WebSphere MQ for HP-UX Version 7.0
- IBM WebSphere MQ for IBM i Version 7.0
- IBM WebSphere MQ for Linux Version 7.0
- IBM WebSphere MQ for Solaris Version 7.0
- IBM WebSphere MQ for Windows Version 7.0
- IBM WebSphere MQ for z/OS Version 7.0

MQCMDL_LEVEL_701

Level 701 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 7.0.1
- IBM WebSphere MQ for HP-UX Version 7.0.1
- IBM WebSphere MQ for IBM i Version 7.0.1
- IBM WebSphere MQ for Linux Version 7.0.1
- IBM WebSphere MQ for Solaris Version 7.0.1
- IBM WebSphere MQ for Windows Version 7.0.1
- IBM WebSphere MQ for z/OS Version 7.0.1

MQCMDL_LEVEL_710

Level 710 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 7.1
- IBM WebSphere MQ for HP-UX Version 7.1
- IBM WebSphere MQ for IBM i Version 7.1
- IBM WebSphere MQ for Linux Version 7.1
- IBM WebSphere MQ for Solaris Version 7.1
- IBM WebSphere MQ for Windows Version 7.1
- IBM WebSphere MQ for z/OS Version 7.1

MQCMDL_LEVEL_750

Level 750 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 7.5
- IBM WebSphere MQ for HP-UX Version 7.5
- IBM WebSphere MQ for IBM i Version 7.5
- IBM WebSphere MQ for Linux Version 7.5
- IBM WebSphere MQ for Solaris Version 7.5
- IBM WebSphere MQ for Windows Version 7.5

MQCMDL_LEVEL_800

Level 800 of system control commands.

This value is returned by the following versions:

- IBM WebSphere MQ for AIX Version 8.0
- IBM WebSphere MQ for HP-UX Version 8.0
- IBM WebSphere MQ for IBM i Version 8.0
- IBM WebSphere MQ for Linux Version 8.0
- IBM WebSphere MQ for Solaris Version 8.0
- IBM WebSphere MQ for Windows Version 8.0
- IBM WebSphere MQ for z/OS Version 8.0

► V 8.0.0.2 MQCMDL_LEVEL_801

Level 801 of system control commands.

This value is returned by the following versions:

- IBM MQ for AIX Version 8.0.0.2
- IBM MQ for HP-UX Version 8.0.0.2
- IBM MQ for IBM i Version 8.0.0.2
- IBM MQ for Linux Version 8.0.0.2
- IBM MQ for Solaris Version 8.0.0.2

► V 8.0.0.3 MQCMDL_LEVEL_802

Level 802 of system control commands.

This value is returned by the following versions:

- IBM MQ for AIX Version 8.0.0.3
- IBM MQ for HP-UX Version 8.0.0.3
- IBM MQ for IBM i Version 8.0.0.3
- IBM MQ for Linux Version 8.0.0.3
- IBM MQ for Solaris Version 8.0.0.3
- IBM MQ for Windows Version 8.0.0.3

The set of system control commands that corresponds to a particular value of the *CommandLevel* attribute varies according to the value of the *Platform* attribute; both must be used to decide which system control commands are supported.

To determine the value of this attribute, use the MQIA_COMMAND_LEVEL selector with the MQINQ call.

CommandServerControl (MQLONG)

Specifies whether the command server is to be started when the queue manager starts.

The value can be any of the following values:

MQSVC_CONTROL_MANUAL

The command server is not to be started automatically.

MQSVC_CONTROL_Q_MGR

The command server is to be started automatically when the queue manager starts.

This attribute is not supported on z/OS.

To determine the value of this attribute, use the MQIA_CMD_SERVER_CONTROL selector with the MQINQ call.

ConfigurationEvent (MQLONG)

Controls whether configuration events are generated.

To determine the value of this attribute, use the MQIA_CONFIGURATION_EVENT selector with the MQINQ call.

The value can be any of the following values:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

DeadLetterQName (MQCHAR48)

This is the name of a queue defined on the local queue manager as the dead-letter (undelivered-message) queue. Messages are sent to this queue if they cannot be routed to their correct destination.

For example, messages are put on this queue when:

- A message arrives at a queue manager, destined for a queue that is not yet defined on that queue manager
- A message arrives at a queue manager, but the queue for which it is destined cannot receive it because, possibly:
 - The queue is full
 - Put requests are inhibited
 - The sending node does not have authority to put messages on the queue

Applications can also put messages on the dead-letter queue.

Report messages are treated in the same way as ordinary messages; if the report message cannot be delivered to its destination queue (usually the queue specified by the *ReplyToQ* field in the message descriptor of the original message), the report message is placed on the dead-letter (undelivered-message) queue.

Note: Messages that have passed their expiry time (see [MQMD - Expiry field](#)) are **not** transferred to this queue when they are discarded. However, an expiration report message (MQRO_EXPIRATION) is still generated and sent to the *ReplyToQ* queue, if requested by the sending application.

Messages are not put on the dead-letter (undelivered-message) queue when the application that issued the put request has been notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call (for example, a message put on a local queue for which put requests are inhibited).

Messages on the dead-letter (undelivered-message) queue sometimes have their application message data prefixed with an MQDLH structure. This structure contains extra information that indicates why the message was placed on the dead-letter (undelivered-message) queue. See [“MQDLH - Dead-letter header”](#) on page 333 for more details of this structure.

This queue must be a local queue, with a *Usage* attribute of MQUS_NORMAL.

If a queue manager does not support a dead-letter (undelivered-message) queue, or one has not been defined, the name is all blanks. All IBM MQ queue managers support a dead-letter (undelivered-message) queue, but by default it is not defined.

If the dead-letter (undelivered-message) queue is not defined, full, or unusable for some other reason, a message which would have been transferred to it by a message channel agent is retained instead on the transmission queue.

To determine the value of this attribute, use the MQCA_DEAD_LETTER_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

DefClusterXmitQueueType (MQLONG)

The DefClusterXmitQueueType attribute controls which transmission queue is selected by default by cluster-sender channels to get messages from, to send the messages to cluster-receiver channels.

The values of **DefClusterXmitQueueType** are MQCLXQ_SCTQ or MQCLXQ_CHANNEL.

MQCLXQ_SCTQ

All cluster-sender channels send messages from `SYSTEM.CLUSTER.TRANSMIT.QUEUE`. The `correlID` of messages placed on the transmission queue identifies which cluster-sender channel the message is destined for.

SCTQ is set when a queue manager is defined. This behavior is implicit in versions of IBM WebSphere MQ, earlier than Version 7.5. In earlier versions, the queue manager attribute DefClusterXmitQueueType was not present.

MQCLXQ_CHANNEL

Each cluster-sender channel sends messages from a different transmission queue. Each transmission queue is created as a permanent dynamic queue from the model queue `SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE`.

If the queue manager attribute, DefClusterXmitQueueType, is set to CHANNEL, the default configuration is changed to cluster-sender channels being associated with individual cluster transmission queues. The transmission queues are permanent-dynamic queues created from the model queue `SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE`. Each transmission queue is associated with one cluster-sender channel. As one cluster-sender channel services a cluster transmission queue, the transmission queue contains messages for only one queue manager in one cluster. You can configure clusters so that each queue manager in a cluster contains only one cluster queue. In this case, the message traffic from a queue manager to each cluster queue is transferred separately from messages to other queues.

To query the value, call MQINQ, or send an Inquire Queue Manager (MQCMD_INQUIRE_Q_MGR) PCF command, setting the MQIA_DEF_CLUSTER_XMIT_Q_TYPE selector. To change the value, send a Change Queue Manager (MQCMD_CHANGE_Q_MGR) PCF command, setting the MQIA_DEF_CLUSTER_XMIT_Q_TYPE selector.

Related reference

[Change Queue Manager](#)

[Inquire Queue Manager](#)

“MQINQ - Inquire object attributes” on page 677

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object.

DefXmitQName (MQCHAR48)

This is the name of the transmission queue that is used for the transmission of messages to remote queue managers, if there is no other indication of which transmission queue to use.

If there is no default transmission queue, the name is entirely blank. The initial value of this attribute is blank.

To determine the value of this attribute, use the MQCA_DEF_XMIT_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

DistLists (MQLONG)

This indicates whether the local queue manager supports distribution lists on the MQPUT and MQPUT1 calls. It is one of the following values:

MQDL_SUPPORTED

Distribution lists supported.

MQDL_NOT_SUPPORTED

Distribution lists not supported.

To determine the value of this attribute, use the MQIA_DIST_LISTS selector with the MQINQ call.

DNSGroup (MQCHAR18)

This parameter is no longer used. Refer to [WLM/DNS no longer supported](#).

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQCA_DNS_GROUP selector with the MQINQ call. The length of this attribute is given by MQ_DNS_GROUP_NAME_LENGTH.

DNSWLM (MQLONG)

This parameter is no longer used. Refer to [WLM/DNS no longer supported](#).

The value is one of the following:

MQDNSWLM_YES

This value may be seen on a queue manager migrated from an earlier release. The value is ignored.

MQDNSWLM_NO

This is the only value supported by the queue manager.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_DNS_WLM selector with the MQINQ call.

ExpiryInterval (MQLONG)

This indicates the frequency with which the queue manager scans the queues looking for expired messages. It is either a time interval in seconds in the range 1 through 99 999 999, or the following special value:

MQEXPI_OFF

The queue manager does not scan the queues looking for expired messages.

To determine the value of this attribute, use the MQIA_EXPIRY_INTERVAL selector with the MQINQ call.

This attribute is supported only on z/OS.

IGQPutAuthority (MQLONG)

This attribute applies only if the local queue manager is a member of a queue-sharing group. It indicates the type of authority checking that is performed when the local intra-group queuing agent (IGQ agent) removes a message from the shared transmission queue and places the message on a local queue. The value is one of the following:

MQIGQPA_DEFAULT

The user identifier checked for authorization is the value of the *UserIdentifier* field in the *separate* MQMD that is associated with the message when the message is on the shared transmission queue. This is the user identifier of the program that placed the message on the shared transmission queue, and is usually the same as the user identifier under which the remote queue manager is running.

If the RESLEVEL profile indicates that more than one user identifier is to be checked, the user identifier of the local IGQ agent (*IGQUserId*) is also checked.

MQIGQPA_CONTEXT

The user identifier checked for authorization is the value of the *UserIdentifier* field in the *separate* MQMD that is associated with the message when the message is on the shared transmission queue. This is the user identifier of the program that placed the message on the shared transmission queue, and is usually the same as the user identifier under which the remote queue manager is running.

If the RESLEVEL profile indicates that more than one user identifier is to be checked, the user identifier of the local IGQ agent (*IGQUserId*) and the value of the *UserIdentifier* field in the *embedded* MQMD are also checked. The latter user identifier is usually the user identifier of the application that originated the message.

MQIGQPA_ONLY_IGQ

The user identifier checked for authorization is the user identifier of the local IGQ agent (*IGQUserId*).

If the RESLEVEL profile indicates that more than one user identifier is to be checked, this user identifier is used for all checks.

MQIGQPA_ALTERNATE_OR_IGQ

The user identifier checked for authorization is the user identifier of the local IGQ agent (*IGQUserId*).

If the RESLEVEL profile indicates that more than one user identifier is to be checked, the value of the *UserIdentifier* field in the *embedded* MQMD is also checked. This user identifier is usually the user identifier of the application that originated the message.

To determine the value of this attribute, use the MQIA_IGQ_PUT_AUTHORITY selector with the MQINQ call.

This attribute is supported only on z/OS.

IGQUserId (MQLONG)

This attribute is applicable only if the local queue manager is a member of a queue-sharing group. It specifies the user identifier that is associated with the local intra-group queuing agent (IGQ agent). This identifier is one of the user identifiers that can be checked for authorization when the IGQ agent puts messages on local queues. The actual user identifiers checked depend on the setting of the *IGQPutAuthority* attribute, and on external security options.

If *IGQUserId* is blank, no user identifier is associated with the IGQ agent and the corresponding authorization check is not performed (although other user identifiers might still be checked for authorization).

To determine the value of this attribute, use the MQCA_IGQ_USER_ID selector with the MQINQ call. The length of this attribute is given by MQ_USER_ID_LENGTH.

This attribute is supported only on z/OS.

InhibitEvent (MQLONG)

This controls whether inhibit (Inhibit Get and Inhibit Put) events are generated. The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_INHIBIT_EVENT selector with the MQINQ call.

On z/OS, you cannot use the MQINQ call to determine the value of this attribute.

IntraGroupqueuing (MQLONG)

This attribute applies only if the local queue manager is a member of a queue-sharing group. It indicates whether intra-group queuing is enabled for the queue-sharing group. The value is one of the following:

MQIGQ_DISABLED

All messages destined for other queue managers in the queue-sharing group are transmitted using conventional channels..

MQIGQ_ENABLED

Messages destined for other queue managers in the queue-sharing group are transmitted using the shared transmission queue if the following condition is satisfied:

- The length of the message data plus transmission header does not exceed 63 KB (64 512 bytes).

It is recommended that somewhat more space than the size of MQXQH be allocated for the transmission header; the constant MQ_MSG_HEADER_LENGTH is provided for this purpose.

If this condition is not satisfied, the message is transmitted using conventional channels.

Note: When intra-group queuing is enabled, the order of messages transmitted using the shared transmission queue is not preserved relative to those transmitted using conventional channels.

To determine the value of this attribute, use the MQIA_INTRA_GROUP_queuing selector with the MQINQ call.

This attribute is supported only on z/OS.

IPAddressVersion (MQLONG)

Specifies which IP address version, either IPv4 or IPv6, is used.

This attribute is only relevant for systems that run both IPv4 and IPv6 and only affects channels defined as having a *TransportType* of MQXPY_TCP when one of the following conditions is true:

- The channel's *ConnectionName* is a host name that resolves to both an IPv4 and IPv6 address and its *LocalAddress* parameter is not specified.
- The channel's *ConnectionName* and *LocalAddress* are both host names that resolve to both IPv4 and IPv6 addresses.

The value can be any of the following values:

MQIPADDR_IPv4

IPv4 is used.

MQIPADDR_IPv6

IPv6 is used.

To determine the value of this attribute, use the MQIA_IP_ADDRESS_VERSION selector with the MQINQ call.

ListenerTimer (MQLONG)

This is the time interval (in seconds) between IBM MQ attempts to restart the listener if there has been an APPC or TCP/IP failure. The value must be between 5 and 9999, with a default value of 60.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_LISTENER_TIMER selector with the MQINQ call.

LocalEvent (MQLONG)

This controls whether local error events are generated. The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_LOCAL_EVENT selector with the MQINQ call.

On z/OS, you cannot use the MQINQ call to determine the value of this attribute.

LoggerEvent (MQLONG)

This controls whether recovery log events are generated. The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_LOGGER_EVENT selector with the MQINQ call.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, and Windows.

LUGroupName (MQCHAR8)

This is the generic LU name for the LU 6.2 listener that handles inbound transmissions for the queue-sharing group. If you leave this name blank, you cannot use this listener.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQCA_LU_GROUP_NAME selector with the MQINQ call. The length of this attribute is given by MQ_LU_NAME_LENGTH.

LUName (MQCHAR8)

This is the name of the LU to use for outbound LU 6.2 transmissions. Set this to the same LU that the listener uses for inbound transmissions. If you leave this name blank, the APPC/MVS default LU is used; this is variable, so always set LUName if you are using LU6.2.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQCA_LU_NAME selector with the MQINQ call. The length of this attribute is given by MQ_LU_NAME_LENGTH.

LU62ARMSuffix (MQCHAR2)

This is the suffix of the SYS1.PARMLIB member APPCPMxx, that nominates the LUADD for this channel initiator. The z/OS command SET APPC=xx is issued when ARM restarts the channel initiator. If you leave this name blank, no SET APPC=xx is issued.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQCA_LU62_ARM_SUFFIX selector with the MQINQ call. The length of this attribute is given by MQ_ARM_SUFFIX_LENGTH.

LU62Channels (MQLONG)

This is the maximum number of channels that can be current, or clients that can be connected, that use the LU 6.2 transmission protocol.

The value must be in the range 0 through 9999, with a default value of 200. If you set this to zero, the LU 6.2 transmission protocol is not used.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_LU62_CHANNELS selector with the MQINQ call.

MaxActiveChannels (MQLONG)

This attribute is the maximum number of channels that can be *active* at any time.

The default is the value specified for the MaxChannels attribute.

For z/OS, the value must be in the range 1 through 9 999.

For all other platforms, the value must be in the range 1 through 65 535.

The **MaxActiveChannels** parameter is a queue manager attribute on z/OS only. On the other platforms, **MaxActiveChannels** is an attribute in the `qm.ini` file. See [Configuration file stanzas for distributed queuing](#) for information on how you set the **MaxActiveChannels** attribute on other platforms.

To determine the value of this attribute, use the MQIA_ACTIVE_CHANNELS selector with the **MQINQ** call.

Related concepts

[Channel states](#)

MaxChannels (MQLONG)

This attribute is the maximum number of channels that can be *current* (including server-connection channels with connected clients).

For z/OS, the value must be in the range 1 through 9 999, with a default value of 200.

For all other platforms, the value must be in the range 1 through 65 535, with a default value of 100.

A system that is busy serving connections from the network might need a higher number than the default setting. Determine the value that is correct for your environment, ideally by observing the behavior of your system during testing.

The **MaxChannels** parameter is a queue manager attribute on z/OS only. On the other platforms, **MaxChannels** is an attribute in the `qm.ini` file. See [Configuration file stanzas for distributed queuing](#) for information on how you set the **MaxChannels** attribute on other platforms.

To determine the value of this attribute, use the MQIA_MAX_CHANNELS selector with the MQINQ call.

Related concepts

[Channel states](#)

MaxHandles (MQLONG)

This is the maximum number of open handles that any one task can use concurrently. Each successful MQOPEN call for a single queue (or for an object that is not a queue) uses one handle. That handle becomes available for reuse when the object is closed. However, when a distribution list is opened, each queue in the distribution list is allocated a separate handle, and so that MQOPEN call uses as many handles as there are queues in the distribution list. This must be taken into account when deciding on a suitable value for *MaxHandles*.

The MQPUT1 call performs an MQOPEN call as part of its processing; as a result, MQPUT1 uses as many handles as MQOPEN would, but the handles are used only for the duration of the MQPUT1 call itself.

On z/OS, *task* means a CICS task, an MVS task, or an IMS dependent region.

The value is in the range 1 through 999 999 999. The default value is determined by the environment:

- On z/OS, the default value is 100.
- In all other environments, the default value is 256.

To determine the value of this attribute, use the MQIA_MAX_HANDLES selector with the MQINQ call.

MaxMsgLength (MQLONG)

This is the length of the longest *physical* message that the queue manager can handle. However, because the *MaxMsgLength* queue-manager attribute can be set independently of the *MaxMsgLength* queue attribute, the longest physical message that can be placed on a queue is the lesser of those two values.

If the queue manager supports segmentation, an application can put a *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MQMF_SEGMENTATION_ALLOWED flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but usually resource constraints imposed by the operating system, or by the environment in which the application is running, result in a lower limit.

The lower limit for the *MaxMsgLength* attribute is 32 KB (32 768 bytes). The upper limit is 100 MB (104 857 600 bytes).

To determine the value of this attribute, use the MQIA_MAX_MSG_LENGTH selector with the MQINQ call.

MaxPriority (MQLONG)

This is the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to *MaxPriority* (highest).

To determine the value of this attribute, use the MQIA_MAX_PRIORITY selector with the MQINQ call.

MaxPropertiesLength (MQLONG)

This is used to control the size of the properties that can flow with a message. This includes both the property name in bytes and the size of the property value also in bytes.

To determine the value of this attribute, use the MQIA_MAX_PROPERTIES_LENGTH selector with the MQINQ call.

MaxUncommittedMsgs (MQLONG)

This is the maximum number of uncommitted messages that can exist within a unit of work. The number of uncommitted messages is the sum of the following since the start of the current unit of work:

- Messages put by the application with the MQPMO_SYNCPOINT option
- Messages retrieved by the application with the MQGMO_SYNCPOINT option
- Trigger messages and COA report messages generated by the queue manager for messages put with the MQPMO_SYNCPOINT option
- COD report messages generated by the queue manager for messages retrieved with the MQGMO_SYNCPOINT option

The following are *not* counted as uncommitted messages:

- Messages put or retrieved by the application outside a unit of work
- Trigger messages or COA/COD report messages generated by the queue manager as a result of messages put or retrieved outside a unit of work
- Expiration report messages generated by the queue manager (even if the call causing the expiration report message specified MQGMO_SYNCPOINT)
- Event messages generated by the queue manager (even if the call causing the event message specified MQPMO_SYNCPOINT or MQGMO_SYNCPOINT)

Note:

1. Exception report messages are generated by the Message Channel Agent (MCA), or by the application, and are treated in the same way as ordinary messages put or retrieved by the application.
2. When a message or segment is put with the MQPMO_SYNCPOINT option, the number of uncommitted messages is incremented by one regardless of how many physical messages actually result from the put. (More than one physical message might result if the queue manager must subdivide the message or segment.)
3. When a distribution list is put with the MQPMO_SYNCPOINT option, the number of uncommitted messages is incremented by one *for each physical message that is generated*. This can be as small as one, or as great as the number of destinations in the distribution list.

The lower limit for this attribute is 1; the upper limit is 999 999 999. The default value is 10000.

To determine the value of this attribute, use the MQIA_MAX_UNCOMMITTED_MSGS selector with the MQINQ call.

MQIAccounting (MQLONG)

This controls the collection of accounting information for MQI data.

The value is one of the following:

MQMON_ON

Collect API accounting data.

MQMON_OFF

Do not collect API accounting data. This is the default value.

If you set the queue manager attribute ACCTCONO to ENABLED, this value might be overridden for individual connections using the Options field in the MQCNO structure. Changes to this value are only effective for connections to the queue manager that occur after the change to the attribute.

This attribute is supported only on IBM i, UNIX systems, and Windows.

To determine the value of this attribute, use the MQIA_ACCOUNTING_MQI selector with the MQINQ call.

MQIStatistics (MQLONG)

This controls the collection of statistics monitoring information for the queue manager.

The value is one of the following:

MQMON_ON

Collect MQI statistics.

MQMON_OFF

Do not collect MQI statistics. This is the default value.

This attribute is supported only on IBM i, UNIX and Linux systems, and Windows.

To determine the value of this attribute, use the MQIA_STATISTICS_MQI selector with the MQINQ call.

MsgMarkBrowseInterval (MQLONG)

Time interval in milliseconds after which the queue manager can automatically remove the mark from browse messages.

This is a time interval (in milliseconds) after which the queue manager can automatically remove the mark from browse messages.

This attribute describes the time interval for which messages that have been marked as browsed by a call to MQGET, using the get message option MQGMO_MARK_BROWSE_CO_OP, are expected to remain marked as browsed.

The queue manager might automatically unmark browsed messages that have been marked as browsed for the cooperating set of handles when they have been marked for more than this approximate interval.

This does not affect the state of any message marked as browse, that was obtained by a call to MQGET, using the get message option MQGMO_MARK_BROWSE_HANDLE.

The maximum value is 999 999 999 and the default value is 5000. A special value of -1 for *MsgMarkBrowseInterval* represents an unlimited time interval.



Attention: This value should not be below the default of 5000.

To determine the value of this attribute, use the MQIA_MSG_MARK_BROWSE_INTERVAL selector with the MQINQ call.

OutboundPortMax (MQLONG)

This is the highest port number in the range, defined by OutboundPortMin and OutboundPortMax, of port numbers to be used to bind outgoing channels.

The value is an integer in the range 0 through 65535, and must be equal to or greater than the OutboundPortMin value. The default value is 0.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_OUTBOUND_PORT_MAX selector with the MQINQ call.

OutboundPortMin (MQLONG)

This is the lowest port number in the range, defined by OutboundPortMin and OutboundPortMax, of port numbers to be used to bind outgoing channels.

The value is an integer in the range 0 through 65535, and must be equal to or less than the OutboundPortMax value. The default value is 0.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_OUTBOUND_PORT_MIN selector with the MQINQ call.

PerformanceEvent (MQLONG)

This controls whether performance-related events are generated. It is one of the following values:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_PERFORMANCE_EVENT selector with the MQINQ call.

Platform (MQLONG)

This indicates the operating system on which the queue manager is running:

MQPL_AIX

AIX (same value as MQPL_UNIX).

V 8.0.0.2 MQPL_APPLIANCE

IBM MQ Appliance

MQPL_MVS

z/OS (same value as MQPL_ZOS).

MQPL_NSK

HP Integrity NonStop Server.

MQPL_OS390

z/OS (same value as MQPL_ZOS).

MQPL_OS400

IBM i.

MQPL_UNIX

UNIX systems.

MQPL_WINDOWS_NT

Windows systems.

MQPL_ZOS

z/OS.

To determine the value of this attribute, use the MQIA_PLATFORM selector with the MQINQ call.

PubSubNPInputMsg (MQLONG)

Whether to discard or keep an undelivered input message.

The value is one of the following:

MQUNDELIVERED_DISCARD

Non-persistent input messages may be discarded if they cannot be processed.

This is the default value.

MQUNDELIVERED_KEEP

Non-persistent input messages will not be discarded if they cannot be processed. In this situation the queued publish/subscribe interface will continue to retry the process at appropriate intervals and does not continue processing subsequent messages.

To determine the value of this attribute, use the MQIA_PUBSUB_NP_MSG selector with the MQINQ call.

PubSubNPResponse (MQLONG)

Controls the behavior of undelivered response messages.

The value is one of the following:

MQUNDELIVERED_NORMAL

Non-persistent responses which cannot be placed on the reply queue are put on the dead letter queue, if they cannot be placed on the DLQ then they are discarded.

MQUNDELIVERED_SAFE

Non-persistent responses which cannot be placed on the reply queue are put on the dead letter queue. If the response cannot be set and cannot be placed on the DLQ then the queued publish/subscribe interface will roll back the current operation and then retry at appropriate intervals and does not continue processing subsequent messages.

MQUNDELIVERED_DISCARD

Non-persistent responses are not placed on the reply queue are discarded.

This is the default value for new queue managers.

MQUNDELIVERED_KEEP

Non-persistent responses are not placed on the dead letter queue or discarded. Instead, the queued publish/subscribe interface will back out the current operation and then retry it at appropriate intervals.

To determine the value of this attribute, use the MQIA_PUBSUB_NP_RESP selector with the MQINQ call.

Default value for migrated queue managers.

If the queue manager has been migrated from IBM MQ V6.0, the initial value of this attribute depends on the values of *DiscardNonPersistentResponse* and *DLQNonPersistentResponse* before migration, as shown in the following table.

		DLQNonPersistentResponse		
		Yes	No	Not set
DiscardNonPersistentResponse	Yes	MQUNDELIVERED_NORMAL	MQUNDELIVERED_DISCARD	MQUNDELIVERED_NORMAL
	No	MQUNDELIVERED_SAFE	MQUNDELIVERED_KEEP	MQUNDELIVERED_SAFE
	Not set	If SyncPointPersistent = No, MQUNDELIVERED_SAFE else MQUNDELIVERED_NORMAL	If SyncPointPersistent = No, MQUNDELIVERED_KEEP else MQUNDELIVERED_DISCARD	If SyncPointPersistent = No, MQUNDELIVERED_SAFE else MQUNDELIVERED_NORMAL

PubSubMaxMsgRetryCount (MQLONG)

The number of retries when processing a failed command message under syncpoint.

The value is one of the following:

0 - 999 999 999

The default value is 5.

To determine the value of this attribute, use the MQIA_PUBSUB_MAXMSG_RETRY_COUNT selector with the MQINQ call.

PubSubSyncPoint (MQLONG)

Whether only persistent messages or all messages are processed under syncpoint.

The value is one of the following:

MQSYNCPOINT_IFPER

This makes the queued publish/subscribe interface receive non-persistent messages outside syncpoint. If the daemon receives a publication outside syncpoint, the daemon forwards the publication to subscribers known to it outside syncpoint.

This is the default value.

MQSYNCPOINT_YES

This makes the queued publish/subscribe interface receive all messages under syncpoint.

To determine the value of this attribute, use the MQIA_PUBSUB_SYNC_PT selector with the MQINQ call.

PubSubMode (MQLONG)

Whether the publish/subscribe engine and the queued publish/subscribe interface are running, therefore allowing applications to publish/subscribe by using the application programming interface and the queues that are being monitored by the queued publish/subscribe interface.

The value is one of the following:

MQPSM_COMPAT

The publish/subscribe engine is running. It is therefore possible to publish/subscribe by using the application programming interface. The queued publish/subscribe interface is not running, therefore any message that is put to the queues that are monitored by the queued publish/subscribe interface is not acted on. This setting is used for compatibility with WebSphere Message Broker V6 or earlier versions using this queue manager, because it must read the same queues from which the queued publish/subscribe interface normally reads.

MQPSM_DISABLED

The publish/subscribe engine and the queued publish/subscribe interface are not running. It is therefore not possible to publish/subscribe by using the application programming interface. Any publish/subscribe messages that are put to the queues that are monitored by the queued publish/subscribe interface are not acted on.

MQPSM_ENABLED

The publish/subscribe engine and the queued publish/subscribe interface are running. It is therefore possible to publish/subscribe by using the application programming interface and the queues that are being monitored by the queued publish/subscribe interface. This is the queue manager's initial default value.

To determine the value of this attribute, use the MQIA_PUBSUB_MODE selector with the MQINQ call.

QMGrDesc (MQCHAR64)

Use this field for a commentary describing the queue manager. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

- On z/OS, the default value is the product name and version number.
- In all other environments, the default value is blanks.

To determine the value of this attribute, use the MQCA_Q_MGR_DESC selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_DESC_LENGTH.

QMGrIdentifier (MQCHAR48)

This is an internally-generated unique name for the queue manager.

To determine the value of this attribute, use the MQCA_Q_MGR_IDENTIFIER selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_IDENTIFIER_LENGTH.

This attribute is supported in the following environments: AIX, HP-UX, z/OS, IBM i, Solaris, Linux, Windows, plus IBM MQ clients connected to these systems.

QMGrName (MQCHAR48)

This is the name of the local queue manager, that is, the name of the queue manager to which the application is connected.

The first 12 characters of the name are used to construct a unique message identifier (see [MQMD - MsgId](#) field). Queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, in order for message identifiers to be unique in the queue-manager network.

On z/OS, the name is the same as the subsystem name, which is limited to 4 nonblank characters.

To determine the value of this attribute, use the MQCA_Q_MGR_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_NAME_LENGTH.

QSGName (MQCHAR4)

This is the name of the queue-sharing group to which the local queue manager belongs. If the local queue manager does not belong to a queue-sharing group, the name is blank.

To determine the value of this attribute, use the MQCA_QSG_NAME selector with the MQINQ call. The length of this attribute is given by MQ_QSG_NAME_LENGTH.

This attribute is supported only on z/OS.

QueueAccounting (MQLONG)

This controls the collection of accounting information for queues.

The value is one of the following:

MQMON_NONE

Do not collect accounting data for queues, regardless of the setting of the queue accounting attribute ACCTQ. This is the default value.

MQMON_OFF

Do not collect accounting data for queues that specify QMGR in the ACCTQ queue attribute.

MQMON_ON

Collect accounting data for queues that specify QMGR in the ACCTQ queue attribute.

Changes to this value are only effective for connections to the queue manager that occur after the change to the attribute.

To determine the value of this attribute, use the MQIA_ACCOUNTING_Q selector with the MQINQ call.

QueueMonitoring (MQLONG)

This specifies the default setting for online monitoring of queues.

If the *QueueMonitoring* queue attribute is set to MQMON_Q_MGR, this attribute specifies the value which is assumed by the channel. The value can be:

MQMON_OFF

Online monitoring data collection is turned off. This is the queue manager's initial default value.

MQMON_NONE

Online monitoring data collection is turned off for queues regardless of the setting of their *QueueMonitoring* attribute.

MQMON_LOW

Online monitoring data collection is turned on, with a low ratio of data collection.

MQMON_MEDIUM

Online monitoring data collection is turned on, with a moderate ratio of data collection.

MQMON_HIGH

Online monitoring data collection is turned on, with a high ratio of data collection.

To determine the value of this attribute, use the MQIA_MONITORING_Q selector with the MQINQ call.

QueueStatistics (MQLONG)

This controls the collection of statistics data for queues.

It is one of the following values:

MQMON_NONE

Do not collect queue statistics for queues, regardless of the setting of the *QueueStatistics* queue attribute. This is the default value.

MQMON_OFF

Do not collect statistics data for queues that specify Queue Manager in the *QueueStatistics* queue attribute.

MQMON_ON

Collect statistics data for queues that specify Queue Manager in the *QueueStatistics* queue attribute.

To determine the value of this attribute, use the MQIA_STATISTICS_Q selector with the MQINQ call.

ReceiveTimeout (MQLONG)

This specifies how long a TCP/IP channel waits to receive data, including heartbeats, from its partner before returning to the inactive state. It applies only to message channels and not to MQI channels.

The exact meaning of the ReceiveTimeout is altered by the value specified in ReceiveTimeoutType. ReceiveTimeoutType can be set to one of the following:

- MQRCVTIME_EQUAL - this value is the number in seconds for the channel to wait. Specify a value in the range 0 - 999999.
- MQRCVTIME_ADD - this value is the number in seconds to add to the negotiated HBINT, and it determines how long a channel waits. Specify a value in the range 1 - 999999.
- MQRCVTIME_MULTIPLY - this value is a multiplier to apply to the negotiated HBINT. Specify a value of 0 or a value in the range 2 - 99.

The default value is 0.

Set ReceiveTimeoutType to MQRCVTIME_MULTIPLY or MQRCVTIME_EQUAL, and ReceiveTimeout to 0, to stop a channel from timing out its wait to receive data from its partner.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_RECEIVE_TIMEOUT selector with the MQINQ call.

ReceiveTimeoutMin (MQLONG)

This is the minimum time, in seconds, that a TCP/IP channel waits to receive data, including heartbeats, from its partner, before returning to the inactive state.

It applies only to message channels, not to MQI channels. The value must be in the range 0 through 999999, with a default of 0.

If you use ReceiveTimeoutType to specify that the TCP/IP channel wait time is to be calculated relative to the negotiated value of HBINT, and the resultant value is less than the value of this parameter, this value is used instead.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_RECEIVE_TIMEOUT_MIN selector with the MQINQ call.

ReceiveTimeoutType (MQLONG)

This is the qualifier, applied to ReceiveTimeout to define how long a TCP/IP channel waits to receive data, including heartbeats, from its partner, before returning to the inactive state. It applies only to message channels, not to MQI channels.

The value is one of the following:

MQRCVTIME_MULTIPLY

ReceiveTimeout is a multiplier to apply to the negotiated HBINT value to determine how long a channel waits. This is the default value.

MQRCVTIME_ADD

ReceiveTimeout is a value, in seconds, to add to the negotiated HBINT value to determine how long a channel waits.

MQRCVTIME_EQUAL

ReceiveTimeout is a value, in seconds, that the channel waits.

To stop a channel timing out its wait to receive data from its partner, set ReceiveTimeoutType to MQRCVTIME_MULTIPLY or MQRCVTIME_EQUAL, and ReceiveTimeout to 0.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_RECEIVE_TIMEOUT_TYPE selector with the MQINQ call.

RemoteEvent (MQLONG)

This controls whether remote error events are generated. It is one of the following values:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_REMOTE_EVENT selector with the MQINQ call.

RepositoryName (MQCHAR48)

This is the name of a cluster for which this queue manager provides a repository-manager service. If the queue manager provides this service for more than one cluster, *RepositoryNameList* specifies the name of a namelist object that identifies the clusters, and *RepositoryName* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows, and z/OS.

To determine the value of this attribute, use the MQCA_REPOSITORY_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_NAME_LENGTH.

RepositoryNameList (MQCHAR48)

This is the name of a namelist object that contains the names of clusters for which this queue manager provides a repository-manager service. If the queue manager provides this service for only one cluster, the namelist object contains only one name. Alternatively, *RepositoryName* can be used to specify the name of the cluster, in which case *RepositoryNameList* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

This attribute is supported only on AIX, HP-UX, IBM i, Linux, Solaris, Windows, and z/OS.

To determine the value of this attribute, use the MQCA_REPOSITORY_NAMELIST selector with the MQINQ call. The length of this attribute is given by MQ_NAMELIST_NAME_LENGTH.

ScyCase(MQCHAR8)

Specifies whether the queue manager supports security profile names in mixed case, or in uppercase only.

The value is one of the following:

MQSCYC_UPPER

Security profile names must be in uppercase.

MQSCYC_MIXED

Security profile names can be in uppercase or in mixed case.

Changes to this attribute take effect when a Refresh Security command is run with *SecurityType* (MQSECTYPE_CLASSES) specified.

This attribute is supported only on z/OS.

To determine the value of this attribute, use the MQIA_SECURITY_CASE selector with the MQINQ call.

SharedQMgrName (MQLONG)

This specifies whether the *ObjectQmgrName* should be used or treated as the local queue manager on an MQOPEN call, for a shared queue, when the *ObjectQmgrName* is that of another queue manager in the queue-sharing group.

The value can be any of the following values:

MQSQQM_USE

ObjectQmgrName is used and the appropriate transmission queue is opened.

MQSQQM_IGNORE

If the target queue is shared, and the *ObjectQmgrName* is that of a queue manager in the same queue-sharing group, the open is performed locally.

This attribute is valid only on z/OS.

To determine the value of this attribute, use the MQIA_SHARED_Q_Q_MGR_NAME selector with the MQINQ call.

SPLCAP

Indicates whether security capabilities of IBM MQ Advanced Message Security are available for a queue manager.

MQCAP_SUPPORTED

This is the default value if the IBM MQ AMS component is installed for the installation that the queue manager is running under.

MQCAP_NOT_SUPPORTED

SSLEvent (MQLONG)

This specifies whether SSL events are generated.

It is one of the following values:

MQEVR_ENABLED

Generate SSL events, as follows:

MQRC_CHANNEL_SSL_ERROR

MQEVR_DISABLED

Do not generate SSL events; this is the default value.

To determine the value of this attribute, use the MQIA_SSL_EVENT selector with the MQINQ call.

SSLFIPSRequired (MQLONG)

Note: On UNIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the IBM Crypto for C (ICC) certificate and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the [NIST CMVP modules in process list](#).

This lets you specify that only FIPS-certified algorithms are to be used if the cryptography is executed in IBM MQ, rather than in cryptographic hardware. If cryptographic hardware is configured, the cryptography modules used are those modules provided by the hardware product; these modules might or might not be FIPS-certified to a particular level depending on the hardware product in use.

The value is one of the following values:

MQSSL_FIPS_NO

Use any CipherSpec supported on the platform in use. This value is the default value.

MQSSL_FIPS_YES

Use only FIPS-certified cryptographic algorithms in the CipherSpecs allowed on all SSL connections from and to this queue manager.

This parameter is valid only on UNIX, Linux, Windows, and z/OS platforms.

To determine the value of this attribute, use the MQIA_SSL_FIPS_REQUIRED selector with the MQINQ call.

Related tasks

[Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client](#)

Related reference

[Federal Information Processing Standards \(FIPS\) for UNIX, Linux and Windows](#)

SSLKeyResetCount (MQLONG)

This specifies when SSL channel message channel agents (MCAs) that initiate communication reset the secret key used for encryption on the channel.

The value represents the total number of unencrypted bytes that are sent and received on the channel before the secret key is renegotiated. The number of bytes includes control information sent by the MCA.

The value is a number in the range 0 through 999 999 999, with a default value of 0. If you specify an SSL/TLS secret key reset count in the range 1 byte through 32 KB, SSL/TLS channels will use a secret key reset count of 32 KB. This is to avoid the processing cost of excessive key resets which would occur for small SSL/TLS secret key reset values.

The secret key is renegotiated when the total number of unencrypted bytes sent and received by the initiating channel MCA exceeds the specified value, or if channel heartbeats are enabled before data is sent or received following a channel heartbeat, whichever occurs first.

The count of bytes sent and received for renegotiation includes control information sent and received by the channel MCA and is reset whenever a renegotiation occurs.

Use a value of 0 to indicate that secret keys are never renegotiated.

To determine the value of this attribute, use the MQIA_SSL_RESET_COUNT selector with the MQINQ call.

StartStopEvent (MQLONG)

This controls whether start and stop events are generated. The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_START_STOP_EVENT selector with the MQINQ call.

StatisticsInterval (MQLONG)

This specifies how often (in seconds) to write statistics monitoring data to the monitoring queue.

The value is an integer in the range 0 to 604800, with a default value of 1800 (30 minutes).

To determine the value of this attribute, use the MQIA_STATISTICS_INTERVAL selector with the MQINQ call.

SyncPoint (MQLONG)

This indicates whether the local queue manager supports units of work and syncpointing with the MQGET, MQPUT, and MQPUT1 calls.

MQSP_AVAILABLE

Units of work and syncpointing available.

MQSP_NOT_AVAILABLE

Units of work and syncpointing not available.

- On z/OS this value is never returned.

To determine the value of this attribute, use the MQIA_SYNCPOINT selector with the MQINQ call.

TCPChannels (MQLONG)

This is the maximum number of channels that can be current, or clients that can be connected, that use the TCP/IP transmission protocol.

The value must be in the range 0 through 9999, with a default value of 200. If you specify 0, TCP/IP is not used.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_TCP_CHANNELS selector with the MQINQ call.

TCPKeepAlive (MQLONG)

This specifies whether to use TCP KEEPALIVE to check that the other end of the connection is still available. If it is not available, the channel is closed.

The value is one of the following:

MQTCPKEEP_YES

Use TCP KEEPALIVE as specified in the TCP profile configuration data set. If you specify the channel attribute KeepAliveInterval (KAINT), the value to which it is set is used.

MQTCPKEEP_NO

Do not use TCP KEEPALIVE. This is the default value.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_TCP_KEEP_ALIVE selector with the MQINQ call.

TCPName (MQCHAR8)

This is the name of either the only or preferred TCP/IP stack that will be used, depending on the value of TCPStackType. This parameter is only applicable in CINET multiple stack environments. The default value is TCPIP.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQCA_TCP_NAME selector with the MQINQ call. The length of this attribute is given by MQ_TCP_NAME_LENGTH.

TCPStackType (MQLONG)

This specifies whether the channel initiator can use only the TCP/IP stack specified in TCPName, or can optionally bind to any selected TCP/IP stack. This parameter is only applicable in CINET multiple stack environments.

The value is one of the following:

MQTCPSTACK_SINGLE

The channel initiator can use only the TCP/IP address spaces named in TCPName. This is the default value.

MQTCPSTACK_MULTIPLE

The channel initiator can use any TCP/IP address space available to it. It defaults to the one specified in TCPName if no other is specified for a channel or listener.

This attribute is supported on z/OS only.

To determine the value of this attribute, use the MQIA_TCP_STACK_TYPE selector with the MQINQ call.

TraceRouteRecording (MQLONG)

This controls the recording of trace- route information.

The value is one of the following:

MQRECORDING_DISABLED

No appending to trace- route messages allowed.

MQRECORDING_Q

Put trace- route messages to fixed named queue.

MQRECORDING_MSG

Put trace- route messages to a queue determined using the message itself. This is the default value

To determine the value of this attribute, use the MQIA_TRACE_ROUTE_RECORDING selector with the MQINQ call.

TriggerInterval (MQLONG)

This is a time interval (in milliseconds) used to restrict the number of trigger messages. This is relevant only when the *TriggerType* is MQTT_FIRST. In this case trigger messages are usually generated only when a suitable message arrives on the queue, and the queue was previously empty. Under certain circumstances, however, an additional trigger message can be generated with MQTT_FIRST triggering even if the queue was not empty. These additional trigger messages are not generated more often than every *TriggerInterval* milliseconds.

For more information on triggering, see [Triggering channels](#).

The value is not less than 0 and not greater than 999 999 999. The default value is 999 999 999.

To determine the value of this attribute, use the MQIA_TRIGGER_INTERVAL selector with the MQINQ call.

TriggerInterval (MQLONG)

This is a time interval (in milliseconds) used to restrict the number of trigger messages. This is relevant only when the *TriggerType* is MQTT_FIRST. In this case trigger messages are usually generated only when a suitable message arrives on the queue, and the queue was previously empty. Under certain circumstances, however, an additional trigger message can be generated with MQTT_FIRST triggering even if the queue was not empty. These additional trigger messages are not generated more often than every *TriggerInterval* milliseconds.

For more information on triggering, see [Triggering channels](#).

The value is not less than 0 and not greater than 999 999 999. The default value is 999 999 999.

To determine the value of this attribute, use the MQIA_TRIGGER_INTERVAL selector with the MQINQ call.

Version (MQCFST)

This is the version of the IBM MQ code as VVRRMMFF, where:

VV - Version

RR - Release

MM - Maintenance level

FF - Fix level

XrCapability(MQLONG)

This controls whether IBM MQ Telemetry commands are supported by the queue manager.

The value is one of the following:

MQCAP_SUPPORTED

IBM MQ Telemetry component installed and Telemetry commands are supported.

MQCAP_NOT_SUPPORTED

IBM MQ Telemetry component not installed.

This attribute is supported only on IBM i, Unix systems, and Windows.

To determine the value of this attribute, use the MQIA_XR_CAPABILITY selector with the MQINQ call.

Attributes for queues


There are five types of queue definition. Some queue attributes apply to all types of queue; other queue attributes apply only to certain types of queue.

Types of queue

The queue manager supports the following types of queue definition:

Local queue

You can store messages on a local queue.


 On z/OS you can make it a shared or private queue.

A queue is known to a program as *local* if it is owned by the queue manager to which the program is connected. You can get messages from, and put messages on, local queues.

The queue definition object holds the definition information of the queue as well as the physical messages put on the queue.

Local queue manager queue

The queue exists on the local queue manager.

 The queue is known as a private queue on z/OS.

 **Shared queue (z/OS only)**

The queue exists in a shared repository that is accessible to all the queue managers that belong to the queue-sharing group that owns the shared repository.

Applications connected to any queue manager in the queue-sharing group can place messages on and remove messages from queues of this type. Such queues are effectively the same as local queues. The value of the **QType** queue attribute is MQQT_LOCAL.

Applications connected to the local queue manager can place messages on and remove messages from queues of this type. The value of the **QType** queue attribute is MQQT_LOCAL.

Cluster queue

You can store messages on a cluster queue on the queue manager where it is defined. A cluster queue is a queue that is hosted by a cluster queue manager and made available to other queue managers in the cluster. The value of the **QType** queue attribute is MQQT_CLUSTER.

A cluster queue definition is advertised to other queue managers in the cluster. The other queue managers in the cluster can put messages to a cluster queue without needing a corresponding remote-queue definition. A cluster queue can be advertised in more than one cluster by using a cluster namelist.

When a queue is advertised, any queue manager in the cluster can put messages to it. To put a message, the queue manager must find out, from the full repositories, where the queue is hosted. Then it adds some routing information to the message and puts the message on a cluster transmission queue.

A queue manager can store messages for other queue managers in a cluster on multiple transmission queues. You can configure a queue manager to store messages on multiple cluster transmission queues in two different ways. If you set the queue manager attribute **DEFCLXQ** to CHANNEL, a different cluster transmission queue is created automatically from SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE for each cluster-sender channel. If you set the

CLCHNAME transmission queue option to match one or more cluster-senders channel, the queue manager can store messages for the matching channels on that transmission queue.



Attention: If you are using dedicated `SYSTEM.CLUSTER.TRANSMIT.QUEUES` with a queue manager that was upgraded from a version of the product earlier than IBM WebSphere MQ 7.5, ensure that the `SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE` has the [SHARE/NOSHARE](#) option set to **SHARE**.



A cluster queue can be a queue that is shared by members of a queue-sharing group in IBM MQ for z/OS.

Remote queue

A remote queue is not a physical queue; it is the local definition of a queue that exists on a remote queue manager. The local definition of the remote queue contains information that tells the local queue manager how to route messages to the remote queue manager.

Applications connected to the local queue manager can place messages on queues of this type; the messages are placed on the local transmission queue used to route messages to the remote queue manager. Applications cannot remove messages from remote queues. The value of the **QType** queue attribute is `MQQT_REMOTE`.

You can also use a remote queue definition for:

- Reply-queue aliasing

In this case the name of the definition is the name of a reply-to queue. For more information, see [Reply-to queue aliases and clusters](#).

- Queue-manager aliasing

In this case the name of the definition is an alias for a queue manager, and not the name of a queue. For more information, see [Queue manager aliases and clusters](#).

Alias queue

This is not a physical queue; it is an alternative name for a local queue, a shared queue, a cluster queue, or a remote queue. The name of the queue to which the alias resolves is part of the definition of the alias queue.

Applications connected to the local queue manager can place messages on queues of this type; the messages are placed on the queue to which the alias resolves. Applications can remove messages from queues of this type if the alias resolves to a local queue, a shared queue, or a cluster queue that has a local instance. The value of the **QType** queue attribute is `MQQT_ALIAS`.

Model queue

This is not a physical queue; it is a set of queue attributes from which a local queue can be created.

Messages cannot be stored on queues of this type.

Queue limits



From Version 8.0.0, Fix Pack 15, the queue manager limits the maximum queue file size to 2 TB by default.

Queue attributes

Some queue attributes apply to all types of queue; other queue attributes apply only to certain types of queue. The types of queue to which an attribute applies are shown in [Table 561 on page 803](#) and subsequent tables.

Table 561 on page 803 summarizes the attributes that are specific to queues. The attributes are described in alphabetical order.

Note: The names of the attributes shown in this section are descriptive names used with the MQINQ and MQSET calls; the names are the same as for the PCF commands. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) commands](#) for details.

Table 561. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If any other attributes are inquired, the call returns completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE (2068).
If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.
If the cluster queue is opened for inquire alone, or for inquire and output, plus specifying the base queue manager name, the column for local queues applies instead.

Attribute	Description	Local	Model	Alias	Remote	Cluster
AlterationDate	Date when definition was last changed	✓		✓	✓	
AlterationTime	Time when definition was last changed	✓		✓	✓	
BackoutRequeueQName	Excessive backout requeue queue name	✓	✓			
BackoutThreshold	Backout threshold	✓	✓			
BaseQName	Queue name to which alias resolves			✓		
CFStrucName	Coupling-facility structure name	✓	✓			
CLCHNAME	Cluster-sender channel names	✓	✓			
ClusterName	Name of cluster to which queue belongs	✓		✓	✓	✓
ClusterNamelist	Name of namelist object containing names of clusters to which queue belongs	✓		✓	✓	
CLWLQueuePriority	Cluster workload queue priority	✓		✓	✓	✓
CLWLQueueRank	Cluster workload queue rank	✓		✓	✓	✓
CLWLUseQ	Use remote queue	✓				
CreationDate	Date that the queue was created	✓				
CreationTime	Time that the queue was created	✓				
CurrentQDepth	Current queue depth	✓				
DefaultPutResponse	Default put response	✓	✓	✓	✓	
DefBind	Default binding	✓		✓	✓	✓
DefinitionType attribute	Queue definition type	✓	✓			
DefInputOpenOption	Default input open option	✓	✓			
DefPersistence	Default message persistence	✓	✓	✓	✓	✓
DefPriority	Default message priority	✓	✓	✓	✓	✓
DefReadAhead	Default read ahead	✓	✓	✓		
DistLists	Distribution list support	✓	✓			
HardenGetBackout	Whether to maintain an accurate backout count	✓	✓			

Table 561. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If any other attributes are inquired, the call returns completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE (2068).
If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.
If the cluster queue is opened for inquire alone, or for inquire and output, plus specifying the base queue manager name, the column for local queues applies instead.

(continued)

Attribute	Description	Local	Model	Alias	Remote	Cluster
IndexType	Index type	✓	✓			
InhibitGet	Whether get operations for the queue are allowed	✓	✓	✓		
InhibitPut	Whether put operations for the queue are allowed	✓	✓	✓	✓	✓
InitiationQName	Name of initiation queue	✓	✓			
MaxMsgLength	Maximum message length in bytes	✓	✓			
MaxQDepth	Maximum queue depth	✓	✓			
MsgDeliverySequence attribute	Message delivery sequence	✓	✓			
NonPersistentMessage Class	Reliability goal for non-persistent messages	✓	✓			
OpenInputCount	Number of opens for input	✓				
OpenOutputCount	Number of opens for output	✓				
PropertyControl	Property control	✓	✓	✓		
ProcessName	Process name	✓	✓			
QDepthHighEvent attribute	Whether Queue Depth High events are generated	✓	✓			
QDepthHighLimit	High limit for queue depth	✓	✓			
QDepthLowEvent attribute	Whether Queue Depth Low events are generated	✓	✓			
QDepthLowLimit attribute	Low limit for queue depth	✓	✓			
QDepthMaxEvent	Whether Queue Full events are generated	✓	✓			
QDesc	Queue description	✓	✓	✓	✓	✓
QName	Queue name	✓		✓	✓	✓
QServiceInterval	Target for queue service interval	✓	✓			
QServiceIntervalEvent attribute	Whether Service Interval High or Service Interval OK events are generated	✓	✓			
QSGDisp attribute	Queue-sharing group disposition	✓		✓	✓	
QueueAccounting	Queue accounting data collection	✓	✓	✓	✓	✓
QueueMonitoring	Online monitoring data for queues	✓	✓			
QueueStatistics	Queue statistics data collection	✓	✓	✓	✓	✓
QType	Queue type	✓		✓	✓	✓

Table 561. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If any other attributes are inquired, the call returns completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE (2068).
If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.
If the cluster queue is opened for inquire alone, or for inquire and output, plus specifying the base queue manager name, the column for local queues applies instead.

(continued)

Attribute	Description	Local	Model	Alias	Remote	Cluster
RemoteQMGrName	Name of remote queue manager				✓	
RemoteQName	Name of remote queue				✓	
RetentionInterval	Retention interval	✓	✓			
Scope	Whether an entry for the queue also exists in a cell directory	✓		✓	✓	
Shareability	Queue shareability	✓	✓			
StorageClass	Storage class for queue	✓	✓			
TriggerControl	Trigger control	✓	✓			
TriggerData	Trigger data	✓	✓			
TriggerDepth	Trigger depth	✓	✓			
TriggerMsgPriority	Threshold message priority for triggers	✓	✓			
TriggerType	Trigger type	✓	✓			
Usage attribute	Queue usage	✓	✓			
XmitQName	Transmission queue name				✓	

Related concepts

[Cluster queues](#)

[Local queues](#)

AlterationDate (MQCHAR12)

Date when definition was last changed.

Local	Model	Alias	Remote	Cluster
X		X	X	

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes (for example, 1992-09-23↵↵, where ↵↵ represents two blank characters).

The values of certain attributes (for example, *CurrentQDepth*) change as the queue manager operates. Changes to these attributes do not affect *AlterationDate*.

To determine the value of this attribute, use the MQCA_ALTERATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_DATE_LENGTH.

AlterationTime (MQCHAR8)

Time when definition was last changed.

Local	Model	Alias	Remote	Cluster
X		X	X	

This is the time when the definition was last changed. The format of the time is HH.MM.SS using the 24-hour clock, with a leading zero if the hour is less than 10 (for example 09.10.20).

- On z/OS, the time is Greenwich Mean Time (GMT), subject to the system clock being set accurately to GMT.
- In other environments, the time is local time.

The values of certain attributes (for example, *CurrentQDepth*) change as the queue manager operates. Changes to these attributes do not affect *AlterationTime*.

To determine the value of this attribute, use the MQCA_ALTERATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_TIME_LENGTH.

BackoutRequeueQName (MQCHAR48)

This is the excessive backout requeue queue name. Apart from allowing its value to be queried, the queue manager takes no action based on the value of this attribute.

Local	Model	Alias	Remote	Cluster
X	X			

Applications running inside WebSphere Application Server and those that use the IBM MQ Application Server Facilities use this attribute to determine where messages that have been backed out should go. For all other applications, the queue manager takes no action based on the value of the attribute.

IBM MQ classes for JMS uses this attribute to determine where to transfer a message that has already been backed out the maximum number of times as specified by the *BackoutThreshold* attribute.

To determine the value of this attribute, use the MQCA_BACKOUT_REQ_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

BackoutThreshold (MQLONG)

This is the backout threshold. Apart from allowing its value to be queried, the queue manager takes no action based on the value of this attribute.

Local	Model	Alias	Remote	Cluster
X	X			

Applications running inside of WebSphere Application Server and those that use the IBM MQ Application Server Facilities will use this attribute to determine if a message should be backed out. For all other applications, the queue manager takes no action based on the value of the attribute.

IBM MQ classes for JMS uses this attribute to determine how many times to allow a message to be backed out before transferring the message to the queue specified by the *BackoutRequeueQName* attribute.

To determine the value of this attribute, use the MQIA_BACKOUT_THRESHOLD selector with the MQINQ call.

BaseQName (MQCHAR48)

This is the name of a queue that is defined to the local queue manager.

Local	Model	Alias	Remote	Cluster
		X		

(For more information on queue names, see [MQOD - ObjectName field](#).) The queue is one of the following types:

MQQT_LOCAL

Local queue.

MQQT_REMOTE

Local definition of a remote queue.

MQQT_CLUSTER

Cluster queue.

To determine the value of this attribute, use the MQCA_BASE_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

BaseType (MQCFIN)

The type of object to which the alias resolves.

Local	Model	Alias	Remote	Cluster
		X		

It is one of the following values:

MQOT_Q

Base object type is a queue

MQOT_TOPIC

Base object type is a topic

CFStrucName (MQCHAR12)

This is the name of the coupling-facility structure where the messages on the queue are stored. The first character of the name is in the range A through Z, and the remaining characters are in the range A through Z, 0 through 9, or blank.

Local	Model	Alias	Remote	Cluster
X	X			

To get the full name of the structure in the coupling facility, suffix the value of the *QSGName* queue-manager attribute with the value of the *CFStrucName* queue attribute.

This attribute applies only to shared queues; it is ignored if *QSGDisp* does not have the value MQQSGD_SHARED.

To determine the value of this attribute, use the MQCA_CF_STRUC_NAME selector with the MQINQ call. The length of this attribute is given by MQ_CF_STRUC_NAME_LENGTH.

This attribute is supported only on z/OS.

ClusterChannelName (MQCHAR20)

ClusterChannelName is the generic name of the cluster-sender channels that use this queue as a transmission queue. The attribute specifies which cluster-sender channels send messages to a cluster-receiver channel from this cluster transmission queue.

Local	Model	Alias	Remote	Cluster
✓	✓			

The default queue manager configuration is for all cluster-sender channels to send messages from a single transmission queue, `SYSTEM.CLUSTER.TRANSMIT.QUEUE`. The default configuration can be changed by modified by changing the queue manager attribute, **DefClusterXmitQueueType**. The default value of the attribute is `SCTQ`. You can change the value to `CHANNEL`. If you set the **DefClusterXmitQueueType** attribute to `CHANNEL`, each cluster-sender channel defaults to using a specific cluster transmission queue, `SYSTEM.CLUSTER.TRANSMIT.ChannelName`.

You can also set the transmission queue attribute `ClusterChannelName` attribute to a cluster-sender channel manually. Messages that are destined for the queue manager connected by the cluster-sender channel are stored in the transmission queue that identifies the cluster-sender channel. They are not stored in the default cluster transmission queue. If you set the `ClusterChannelName` attribute to blanks, the channel switches to the default cluster transmission queue when the channel restarts. The default queue is either `SYSTEM.CLUSTER.TRANSMIT.ChannelName` or `SYSTEM.CLUSTER.TRANSMIT.QUEUE`, depending on the value of the queue manager **DefClusterXmitQueueType** attribute.

By specifying asterisks, "*", in **ClusterChannelName**, you can associate a transmission queue with a set of cluster-sender channels. The asterisks can be at the beginning, end, or any number of places in the middle of the channel name string. **ClusterChannelName** is limited to a length of 20 characters: `MQ_CHANNEL_NAME_LENGTH`.

ClusterName (MQCHAR48)

This is the name of the cluster to which the queue belongs.

Local	Model	Alias	Remote	Cluster
X		X	X	X

If the queue belongs to more than one cluster, *ClusterNameList* specifies the name of a namelist object that identifies the clusters, and *ClusterName* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank.

To determine the value of this attribute, use the `MQCA_CLUSTER_NAME` selector with the `MQINQ` call. The length of this attribute is given by `MQ_CLUSTER_NAME_LENGTH`.

ClusterNameList (MQCHAR48)

This is the name of a namelist object that contains the names of clusters to which this queue belongs.

Local	Model	Alias	Remote	Cluster
X		X	X	

If the queue belongs to only one cluster, the namelist object contains only one name. Alternatively, *ClusterName* can be used to specify the name of the cluster, in which case *ClusterNameList* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank.

To determine the value of this attribute, use the `MQCA_CLUSTER_NAMELIST` selector with the `MQINQ` call. The length of this attribute is given by `MQ_NAMELIST_NAME_LENGTH`.

CLWLQueuePriority (MQLONG)

This is the cluster workload queue priority, a value in the range 0 through 9 representing the priority of the queue.

Local	Model	Alias	Remote	Cluster
X		X	X	X

For more information, see [Cluster queues](#).

To determine the value of this attribute, use the `MQIA_CLWL_Q_PRIORITY` selector with the `MQINQ` call.

CLWLQueueRank (MQLONG)

This is the cluster workload queue rank, a value in the range 0 through 9 representing the rank of the queue.

Local	Model	Alias	Remote	Cluster
X		X	X	X

For more information, see [Cluster queues](#).

To determine the value of this attribute, use the MQIA_CLWL_Q_RANK selector with the MQINQ call.

CLWLUseQ (MQLONG)

This defines the behavior of an MQPUT when the target queue has both a local instance and at least one remote cluster instance. If the put originates from a cluster channel, this attribute does not apply.

Local	Model	Alias	Remote	Cluster
X				

The value is one of the following:

MQCLWL_USEQ_ANY

Use remote and local queues.

MQCLWL_USEQ_LOCAL

Do not use remote queues.

MQCLWL_USEQ_AS_Q_MGR

Inherit definition from queue manager's MQIA_CLWL_USEQ.

For more information, see [Cluster queues](#).

To determine the value of this attribute, use the MQCA_CLWL_USEQ selector with the MQINQ call. The length of this attribute is given by MQ_CLWL_USEQ_LENGTH.

CreationDate (MQCHAR12)

This is the date when the queue was created.

Local	Model	Alias	Remote	Cluster
X				

The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes (for example, 2013-09-23↵↵, where ↵↵ represents 2 blank characters).

- On IBM i, the creation date of a queue can differ from that of the underlying operating system entity (file or userspace) that represents the queue.

To determine the value of this attribute, use the MQCA_CREATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_CREATION_DATE_LENGTH.

CreationTime (MQCHAR8)

This is the time when the queue was created.

Local	Model	Alias	Remote	Cluster
X				

The format of the time is HH.MM.SS using the 24-hour clock, with a leading zero if the hour is less than 10 (for example 09.10.20).

- On z/OS, the time is Greenwich Mean Time (GMT), subject to the system clock being set accurately to GMT.

- In other environments, the time is local time.
- On IBM i, the creation time of a queue can differ from that of the underlying operating system entity (file or userspace) that represents the queue.

To determine the value of this attribute, use the MQCA_CREATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_CREATION_TIME_LENGTH.

CurrentQDepth (MQLONG)

This is the number of messages currently on the queue.

Local	Model	Alias	Remote	Cluster
X				

It is incremented during an MQPUT call, and during backout of an MQGET call. It is decremented during a nonbrowse MQGET call, and during backout of an MQPUT call. The effect of this is that the count includes messages that have been put on the queue within a unit of work, but that have not yet been committed, even though they are not eligible to be retrieved by the MQGET call. Similarly, it excludes messages that have been retrieved within a unit of work using the MQGET call, but that have yet to be committed.

The count also includes messages that have passed their expiry time but have not yet been discarded, although these messages are not eligible to be retrieved. See [MQMD - Expiry field](#) for more information.

Unit-of-work processing and the segmentation of messages can both cause *CurrentQDepth* to exceed *MaxQDepth*. However, this does not affect the retrievability of the messages; *all* messages on the queue can be retrieved using the MQGET call in the normal way.

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the MQIA_CURRENT_Q_DEPTH selector with the MQINQ call.

DefaultPutResponse (MQLONG)

Specifies the type of response to be used for put operations to the queue when an application specifies MQPMO_RESPONSE_AS_Q_DEF.

Local	Model	Alias	Remote	Cluster
X	X	X	X	

It is one of the following values:

MQPRT_SYNC_RESPONSE

The put operation is issued synchronously, returning a response.

MQPRT_ASYNC_RESPONSE

The put operation is issued asynchronously, returning a subset of MQMD fields.

DefBind (MQLONG)

This is the default binding that is used when MQOO_BIND_AS_Q_DEF is specified on the MQOPEN call and the queue is a cluster queue.

Local	Model	Alias	Remote	Cluster
X		X	X	X

The value is one of the following:

MQBND_BIND_ON_OPEN

Binding fixed by MQOPEN call.

MQBND_BIND_NOT_FIXED

Binding not fixed.

MQBND_BIND_ON_GROUP

Allows an application to request that a group of messages are all allocated to the same destination instance. Because this value is new in IBM WebSphere MQ 7.1, it must not be used if any of the applications opening this queue are connecting to IBM WebSphere MQ 7.0.1 or earlier queue managers.

To determine the value of this attribute, use the MQIA_DEF_BIND selector with the MQINQ call.

DefinitionType (MQLONG)

This indicates how the queue was defined.

Local	Model	Alias	Remote	Cluster
X	X			

The value is one of the following:

MQQDT_PREDEFINED

The queue is a permanent queue created by the system administrator; only the system administrator can delete it.

Predefined queues are created using the DEFINE MQSC command, and can be deleted only by using the DELETE MQSC command. Predefined queues cannot be created from model queues.

Commands can be issued either by an operator, or by an authorized user sending a command message to the command input queue (see [CommandInputQName attribute](#) for more information).

MQQDT_PERMANENT_DYNAMIC

The queue is a permanent queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value MQQDT_PERMANENT_DYNAMIC for the *DefinitionType* attribute.

This type of queue can be deleted using the MQCLOSE call. See [“MQCLOSE - Close object” on page 622](#) for more details.

The value of the *QSGDisp* attribute for a permanent dynamic queue is MQQSGD_Q_MGR.

MQQDT_TEMPORARY_DYNAMIC

The queue is a temporary queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value MQQDT_TEMPORARY_DYNAMIC for the *DefinitionType* attribute.

This type of queue is deleted automatically by the MQCLOSE call when it is closed by the application that created it.

The value of the *QSGDisp* attribute for a temporary dynamic queue is MQQSGD_Q_MGR.

MQQDT_SHARED_DYNAMIC

The queue is a shared permanent queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value MQQDT_SHARED_DYNAMIC for the *DefinitionType* attribute.

This type of queue can be deleted using the MQCLOSE call. See [“MQCLOSE - Close object” on page 622](#) for more details.

The value of the *QSGDisp* attribute for a shared dynamic queue is MQQSGD_SHARED.

This attribute in a model queue definition does not indicate how the model queue was defined, because model queues are always predefined. Instead, the value of this attribute in the model queue is used to determine the *DefinitionType* of each of the dynamic queues created from the model queue definition using the MQOPEN call.

To determine the value of this attribute, use the MQIA_DEFINITION_TYPE selector with the MQINQ call.

DefInputOpenOption (MQLONG)

This is the default way in which to open the queue for input.

Local	Model	Alias	Remote	Cluster
X	X			

It applies if the MQOO_INPUT_AS_Q_DEF option is specified on the MQOPEN call when the queue is opened. The value is one of the following:

MQOO_INPUT_EXCLUSIVE

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open by this or another application for input of any type (MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE).

MQOO_INPUT_SHARED

Open queue to get messages with shared access.

The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with MQOO_INPUT_SHARED, but fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open with MQOO_INPUT_EXCLUSIVE.

To determine the value of this attribute, use the MQIA_DEF_INPUT_OPEN_OPTION selector with the MQINQ call.

DefPersistence (MQLONG)

This is the default persistence of messages on the queue. It applies if MQPER_PERSISTENCE_AS_Q_DEF is specified in the message descriptor when the message is put.

Local	Model	Alias	Remote	Cluster
X	X	X	X	X

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path at the time of the MQPUT or MQPUT1 call. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value is one of the following:

MQPER_PERSISTENT

The message survives system failures and queue manager restarts. Persistent messages cannot be placed on:

- Temporary dynamic queues
- Shared queues that map to a CFSTRUCT object at CFLEVEL(2) or below, or where the CFSTRUCT object is defined as RECOVER(NO).

Persistent messages can be placed on permanent dynamic queues, and predefined queues.

MQPER_NOT_PERSISTENT

The message does not normally survive system failures or queue manager restarts. This applies even if an intact copy of the message is found on auxiliary storage during a queue manager restart.

In the case of shared queues, nonpersistent messages *do* survive restarts of queue managers in the queue-sharing group, but do not survive failures of the coupling facility used to store messages on the shared queues.

Both persistent and nonpersistent messages can exist on the same queue.

To determine the value of this attribute, use the MQIA_DEF_PERSISTENCE selector with the MQINQ call.

DefPriority (MQLONG)

This is the default priority for messages on the queue. This applies if MQPRI_PRIORITY_AS_Q_DEF is specified in the message descriptor when the message is put on the queue.

Local	Model	Alias	Remote	Cluster
X	X	X	X	X

If there is more than one definition in the queue-name resolution path, the default priority for the message is taken from the value of this attribute in the *first* definition in the path at the time of the put operation. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The way in which a message is placed on a queue depends on the value of the queue's *MsgDeliverySequence* attribute:

- If the *MsgDeliverySequence* attribute is MQMDS_PRIORITY, the logical position at which a message is placed on the queue depends on the value of the *Priority* field in the message descriptor.
- If the *MsgDeliverySequence* attribute is MQMDS_FIFO, messages are placed on the queue as though they had a priority equal to the *DefPriority* of the resolved queue, regardless of the value of the *Priority* field in the message descriptor. However, the *Priority* field retains the value specified by the application that put the message. See [MsgDeliverySequence attribute](#) for more information.

Priorities are in the range zero (lowest) through *MaxPriority* (highest); see [MaxPriority attribute](#).

To determine the value of this attribute, use the MQIA_DEF_PRIORITY selector with the MQINQ call.

DefReadAhead (MQLONG)

Specifies the default read ahead behavior for non-persistent messages delivered to the client.

Local	Model	Alias	Remote	Cluster
X	X	X		

DefReadAhead can be set to one of the following values::

MQREADA_NO

Non-persistent messages are not sent ahead to the client before an applications requests them. A maximum of one non-persistent message can be lost if the client ends abnormally.

MQREADA_YES

Non-persistent messages are sent ahead to the client before an application requests them. Non-persistent messages can be lost if the client ends abnormally or if the client does not consume all the messages it is sent.

MQREADA_DISABLED

Read ahead of non-persistent messages in not enabled for this queue. Messages are not sent ahead to the client regardless of whether read ahead is requested by the client application.

To determine the value of this attribute, use the MQIA_DEF_READ_AHEAD selector with the MQINQ call.

DefPResp (MQLONG)

The default put response type (DEFPRESP) attribute defines the value used by applications when the PutResponseType within MQPMO has been set to MQPMO_RESPONSE_AS_Q_DEF. This attribute is valid for all queue types.

Local	Model	Alias	Remote	Cluster
Yes	Yes	Yes	Yes	Yes

The value is one of the following:

SYNC

The put operation is issued synchronously returning a response.

ASYNC

The put operation is issued asynchronously, returning a subset of MQMD fields.

To determine the value of this attribute, use the MQIA_DEF_PUT_RESPONSE_TYPE selector with the MQINQ call.

DistLists (MQLONG)

This indicates whether distribution-list messages can be placed on the queue.

Local	Model	Alias	Remote	Cluster
X	X			

A message channel agent (MCA) sets the attribute to inform the local queue manager whether the queue manager at the other end of the channel supports distribution lists. This latter queue manager (called the *partnering* queue manager) is the one that next receives the message, after it has been removed from the local transmission queue by a sending MCA.

The sending MCA sets the attribute whenever it establishes a connection to the receiving MCA on the partnering queue manager. In this way, the sending MCA can cause the local queue manager to place on the transmission queue only messages that the partnering queue manager can process correctly.

This attribute is primarily for use with transmission queues, but the processing described is performed regardless of the usage defined for the queue (see [Usage attribute](#)).

The value is one of the following:

MQDL_SUPPORTED

Distribution-list messages can be stored on the queue, and transmitted to the partnering queue manager in that form. This reduces the amount of processing required to send the message to multiple destinations.

MQDL_NOT_SUPPORTED

Distribution-list messages cannot be stored on the queue, because the partnering queue manager does not support distribution lists. If an application puts a distribution-list message, and that message is to be placed on this queue, the queue manager splits the distribution-list message and places the individual messages on the queue instead. This increases the amount of processing required to send the message to multiple destinations, but ensures that the messages are processed correctly by the partnering queue manager.

To determine the value of this attribute, use the MQIA_DIST_LISTS selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is not supported on z/OS.

HardenGetBackout (MQLONG)

For each message, a count is kept of the number of times that the message is retrieved by an MQGET call within a unit of work, and that unit of work subsequently backed out.

Local	Model	Alias	Remote	Cluster
X	X			

This count is available in the *BackoutCount* field in the message descriptor after the MQGET call has completed.

The message backout count survives restarts of the queue manager. However, to ensure that the count is accurate, information has to be *hardened* (recorded on disk or other permanent storage device) each time that an MQGET call retrieves a message within a unit of work for this queue. If this is not done, the queue manager fails, and the MQGET call backs out, the count might or might not be incremented.

Hardening information for each MQGET call within a unit of work, however, imposes additional processing cost, so set the *HardenGetBackout* attribute to MQQA_BACKOUT_HARDENED only if it is essential that the count is accurate.

On IBM i, UNIX systems, and Windows, the message backout count is always hardened, regardless of the setting of this attribute.

The following values are possible:

MQQA_BACKOUT_HARDENED

Hardening is used to ensure that the backout count for messages on this queue is accurate.

MQQA_BACKOUT_NOT_HARDENED

Hardening is not used to ensure that the backout count for messages on this queue is accurate. The count might therefore be lower than it should be.

To determine the value of this attribute, use the MQIA_HARDEN_GET_BACKOUT selector with the MQINQ call.

IndexType (MQLONG)

This specifies the type of index that the queue manager maintains for messages on the queue.

Local	Model	Alias	Remote	Cluster
X	X			

The type of index required depends on how the application retrieves messages, and whether the queue is a shared queue or a nonshared queue (see [QSGDisp attribute](#)). The following values are possible for *IndexType*:

MQIT_NONE

No index is maintained by the queue manager for this queue. Use this value for queues that are typically processed sequentially, that is, without using any selection criteria on the MQGET call.

MQIT_MSG_ID

The queue manager maintains an index that uses the message identifiers of the messages on the queue. Use this value queues where the application typically retrieves messages using the message identifier as the selection criterion on the MQGET call.

MQIT_CORREL_ID

The queue manager maintains an index that uses the correlation identifiers of the messages on the queue. Use this value for queues where the application typically retrieves messages using the correlation identifier as the selection criterion on the MQGET call.

MQIT_MSG_TOKEN

Important: This index type should only be used for queues used with the IBM MQ Workflow for z/OS product.

The queue manager maintains an index that uses the message tokens of the messages on the queue for use with the workload manager (WLM) functions of z/OS.

You *must* specify this option for WLM-managed queues; do not specify it for any other type of queue. Also, do not use this value for a queue where an application is not using the z/OS workload manager functions, but is retrieving messages using the message token as a selection criterion on the MQGET call.

MQIT_GROUP_ID

The queue manager maintains an index that uses the group identifiers of the messages on the queue. This value *must* be used for queues where the application retrieves messages using the MQGMO_LOGICAL_ORDER option on the MQGET call.

A queue with this index type cannot be a transmission queue. A shared queue with this index type must be defined to map to a CFSTRUCT object at CFLEVEL(3) or higher.

Note:

1. The physical order of messages on a queue with index type MQIT_GROUP_ID is not defined, as the queue is optimized for efficient retrieval of messages using the MQGMO_LOGICAL_ORDER option on the MQGET call. This means that the physical order of the messages is not typically the order in which the messages arrived on the queue.
2. If an MQIT_GROUP_ID queue has a *MsgDeliverySequence* of MQMDS_PRIORITY, the queue manager uses message priorities 0 and 1 to optimize the retrieval of messages in logical order. As a result, the first message in a group must not have a priority of zero or one; if it does, the message is processed as though it had a priority of two. The *Priority* field in the MQMD structure is not changed.

For more information about message groups, see the description of the group and segment options in [MQGMO - Options field](#).

The index type that should be used in various cases is shown in [Table 562 on page 816](#) and [Table 563 on page 817](#).

Table 562. Suggested or required values of queue index type when MQGMO_LOGICAL_ORDER not specified		
Selection criteria on MQGET call	Index type for nonshared queue	Index type for shared queue
None	Any	Any
Selection using one identifier:		
Message identifier	MQIT_MSG_ID suggested	MQIT_NONE or MQIT_MSG_ID required; MQIT_MSG_ID suggested
Correlation identifier	MQIT_CORREL_ID suggested	MQIT_CORREL_ID required
Group identifier	MQIT_GROUP_ID suggested	MQIT_GROUP_ID required
Selection using two identifiers:		
Message identifier plus correlation identifier	MQIT_MSG_ID or MQIT_CORREL_ID suggested	MQIT_NONE or MQIT_MSG_ID or MQIT_CORREL_ID required (For efficiency, it is suggested that the index type is chosen to match the MQMD field which will have the most distinct keys)
Message identifier plus group identifier	MQIT_MSG_ID or MQIT_GROUP_ID suggested	Not supported

Table 562. Suggested or required values of queue index type when MQGMO_LOGICAL_ORDER not specified (continued)

Selection criteria on MQGET call	Index type for nonshared queue	Index type for shared queue
Correlation identifier plus group identifier	MQIT_CORREL_ID or MQIT_GROUP_ID suggested	Not supported
Selection using three identifiers:		
Message identifier plus correlation identifier plus group identifier	MQIT_MSG_ID or MQIT_CORREL_ID or MQIT_GROUP_ID suggested	Not supported
Selection using group-related criteria:		
Group identifier plus message sequence number	MQIT_GROUP_ID required	MQIT_GROUP_ID required
Message sequence number (must be 1)	MQIT_GROUP_ID required	MQIT_GROUP_ID required
Selection using message token:		
Message token for application use	Do not use MQIT_MSG_TOKEN	
Message token for WLM use	MQIT_MSG_TOKEN required	Not supported

Table 563. Suggested or required values of queue index type when MQGMO_LOGICAL_ORDER specified

Selection criteria on MQGET call	Index type for nonshared queue	Index type for shared queue
None	MQIT_GROUP_ID required	MQIT_GROUP_ID required
Selection using one identifier:		
Message identifier	MQIT_GROUP_ID required	Not supported
Correlation identifier	MQIT_GROUP_ID required	Not supported
Group identifier	MQIT_GROUP_ID required	MQIT_GROUP_ID required
Selection using two identifiers:		
Message identifier plus correlation identifier	MQIT_GROUP_ID required	Not supported
Message identifier plus group identifier	MQIT_GROUP_ID required	Not supported
Correlation identifier plus group identifier	MQIT_GROUP_ID required	Not supported
Selection using three identifiers:		
Message identifier plus correlation identifier plus group identifier	MQIT_GROUP_ID required	Not supported

To determine the value of this attribute, use the MQIA_INDEX_TYPE selector with the MQINQ call.

This attribute is supported only on z/OS.

InhibitGet (MQLONG)

This controls whether get operations for this queue are allowed.

Local	Model	Alias	Remote	Cluster
X	X	X		

If the queue is an alias queue, get operations must be allowed for both the alias and the base queue at the time of the get operation, for the MQGET call to succeed. The value is one of the following:

MQQA_GET_INHIBITED

Get operations are inhibited.

MQGET calls fail with reason code MQRC_GET_INHIBITED. This includes MQGET calls that specify MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT.

Note: If an MQGET call operating within a unit of work completes successfully, changing the value of the *InhibitGet* attribute subsequently to MQQA_GET_INHIBITED does not prevent the unit of work being committed.

MQQA_GET_ALLOWED

Get operations are allowed.

To determine the value of this attribute, use the MQIA_INHIBIT_GET selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InhibitPut (MQLONG)

This controls whether put operations for this queue are allowed.

Local	Model	Alias	Remote	Cluster
X	X	X	X	X

If there is more than one definition in the queue-name resolution path, put operations must be allowed for every definition in the path (including any queue-manager alias definitions) at the time of the put operation, for the MQPUT or MQPUT1 call to succeed. The value is one of the following:

MQQA_PUT_INHIBITED

Put operations are inhibited.

MQPUT and MQPUT1 calls fail with reason code MQRC_PUT_INHIBITED.

Note: If an MQPUT call operating within a unit of work completes successfully, changing the value of the *InhibitPut* attribute subsequently to MQQA_PUT_INHIBITED does not prevent the unit of work being committed.

MQQA_PUT_ALLOWED

Put operations are allowed.

To determine the value of this attribute, use the MQIA_INHIBIT_PUT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InitiationQName (MQCHAR48)

This is the name of a queue defined on the local queue manager; the queue must be of type MQQT_LOCAL.

Local	Model	Alias	Remote	Cluster
X				

The queue manager sends a trigger message to the initiation queue when application start-up is required as a result of a message arriving on the queue to which this attribute belongs. The initiation queue must be monitored by a trigger monitor application that starts the appropriate application after receipt of the trigger message.

To determine the value of this attribute, use the MQCA_INITIATION_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

MaxMsgLength (MQLONG)

This is an upper limit for the length of the longest *physical* message that can be placed on the queue.

Local	Model	Alias	Remote	Cluster
X	X			

However, because the *MaxMsgLength* queue attribute can be set independently of the *MaxMsgLength* queue-manager attribute, the actual upper limit for the length of the longest physical message that can be placed on the queue is the lesser of those two values.

If the queue manager supports segmentation, it is possible for an application to put a *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MQMF_SEGMENTATION_ALLOWED flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but usually resource constraints imposed by the operating system, or by the environment in which the application is running, result in a lower limit.

An attempt to place on the queue a message that is too long fails with one of the following reason codes:

- MQRC_MSG_TOO_BIG_FOR_Q if the message is too big for the queue
- MQRC_MSG_TOO_BIG_FOR_Q_MGR if the message is too big for the queue manager, but not too big for the queue

The lower limit for the *MaxMsgLength* attribute is zero; the upper limit is 100 MB (104 857 600 bytes).

For more information, see [MQPUT - BufferLength parameter](#).

To determine the value of this attribute, use the MQIA_MAX_MSG_LENGTH selector with the MQINQ call.

MaxQDepth (MQLONG)

This is the defined upper limit for the number of physical messages that can exist on the queue at any one time.

Local	Model	Alias	Remote	Cluster
X	X			

An attempt to put a message on a queue that already contains *MaxQDepth* messages fails with reason code MQRC_Q_FULL.

Unit-of-work processing and the segmentation of messages can both cause the actual number of physical messages on the queue to exceed *MaxQDepth*. However, this does not affect the retrievability of the messages; *all* messages on the queue can be retrieved using the MQGET call.

The value of this attribute is zero or greater. The upper limit is determined by the environment:

- On AIX, HP-UX, z/OS, Solaris, Linux, and Windows, the value cannot exceed 999 999 999.
- On IBM i, the value cannot exceed 640 000.

Note: The storage space available to the queue might be exhausted even if there are fewer than *MaxQDepth* messages on the queue.

To determine the value of this attribute, use the MQIA_MAX_Q_DEPTH selector with the MQINQ call.

MsgDeliverySequence (MQLONG)

Local	Model	Alias	Remote	Cluster
X	X			

This determines the order in which the MQGET call returns messages to the application :

MQMDS_FIFO

Messages are returned in FIFO order (first in, first out).

An MQGET call returns the *first* message that satisfies the selection criteria specified on the call, regardless of the priority of the message.

MQMDS_PRIORITY

Messages are returned in priority order.

An MQGET call returns the *highest-priority* message that satisfies the selection criteria specified on the call. Within each priority level, messages are returned in FIFO order (first in, first out).

- On z/OS, if the queue has an *IndexType* of MQIT_GROUP_ID, the *MsgDeliverySequence* attribute specifies the order in which message groups are returned to the application. The particular sequence in which the groups are returned is determined by the position or priority of the first message in each group. The physical order of messages on the queue is not defined, as the queue is optimized for efficient retrieval of messages using the MQGMO_LOGICAL_ORDER option on the MQGET call.
- On z/OS, if *IndexType* is MQIT_GROUP_ID and *MsgDeliverySequence* is MQMDS_PRIORITY, the queue manager uses message priorities zero and one to optimize the retrieval of messages in logical order. As a result, the first message in a group must not have a priority of zero or one; if it does, the message is processed as though it had a priority of two. The *Priority* field in the MQMD structure is not changed.

If the relevant attributes are changed while there are messages on the queue, the delivery sequence is as follows:

- The order in which messages are returned by the MQGET call is determined by the values of the *MsgDeliverySequence* and *DefPriority* attributes in force for the queue at the time that the message arrives on the queue:
 - If *MsgDeliverySequence* is MQMDS_FIFO when the message arrives, the message is placed on the queue as though its priority were *DefPriority*. This does not affect the value of the *Priority* field in the message descriptor of the message; that field retains the value it had when the message was first put.
 - If *MsgDeliverySequence* is MQMDS_PRIORITY when the message arrives, the message is placed on the queue at the place appropriate to the priority given by the *Priority* field in the message descriptor.

If the value of the *MsgDeliverySequence* attribute is changed while there are messages on the queue, the order of the messages on the queue is not changed.

If the value of the *DefPriority* attribute is changed while there are messages on the queue, the messages are not necessarily delivered in FIFO order, even though the *MsgDeliverySequence* attribute is set to MQMDS_FIFO; those that were placed on the queue at the higher priority are delivered first.

To determine the value of this attribute, use the MQIA_MSG_DELIVERY_SEQUENCE selector with the MQINQ call.

NonPersistentMessageClass (MQLONG)

The reliability goal for nonpersistent messages.

Local	Model	Alias	Remote	Cluster
X	X			

This specifies the circumstances under which nonpersistent messages put on this queue are discarded:

MQNPM_CLASS_NORMAL

Nonpersistent messages are limited to the lifetime of the queue manager session; the messages are discarded in the event of a queue manager restart. This is valid only for non-shared queues, and is the default value.

MQNPM_CLASS_HIGH

The queue manager attempts to retain nonpersistent messages for the lifetime of the queue. Nonpersistent messages might still be lost in the event of a failure. This value is enforced for shared queues.

To determine the value of this attribute, use the MQIA_NPM_CLASS selector with the MQINQ call.

OpenInputCount (MQLONG)

This is the number of handles that are currently valid for removing messages from the queue by means of the MQGET call.

Local	Model	Alias	Remote	Cluster
X				

It is the total number of such handles known to the *local* queue manager. If the queue is a shared queue, the count does not include opens for input that were performed for the queue at other queue managers in the queue-sharing group to which the local queue manager belongs.

The count includes handles where an alias queue that resolves to this queue was opened for input. The count does not include handles where the queue was opened for actions that did not include input (for example, a queue opened only for browse).

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the MQIA_OPEN_INPUT_COUNT selector with the MQINQ call.

OpenOutputCount (MQLONG)

This is the number of handles that are currently valid for adding messages to the queue by means of the MQPUT call.

Local	Model	Alias	Remote	Cluster
X				

It is the total number of such handles known to the *local* queue manager; it does not include opens for output that were performed for this queue at remote queue managers. If the queue is a shared queue, the count does not include opens for output that were performed for the queue at other queue managers in the queue-sharing group to which the local queue manager belongs.

The count includes handles where an alias queue that resolves to this queue was opened for output. The count does not include handles where the queue was opened for actions that did not include output (for example, a queue opened only for inquire).

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the MQIA_OPEN_OUTPUT_COUNT selector with the MQINQ call.

ProcessName (MQCHAR48)

This is the name of a process object that is defined on the local queue manager. The process object identifies a program that can service the queue.

Local	Model	Alias	Remote	Cluster
X	X			

To determine the value of this attribute, use the MQCA_PROCESS_NAME selector with the MQINQ call. The length of this attribute is given by MQ_PROCESS_NAME_LENGTH.

PropertyControl (MQLONG)

Specifies how message properties are handled for messages that are retrieved from queues using the MQGET call with the MQGMO_PROPERTIES_AS_Q_DEF option.

Local	Model	Alias	Remote	Cluster
X	X	X		

The value is one of the following:

MQPROP_ALL

All properties of the message are included with the message when it is delivered to the application. The properties, except those in the message descriptor (or extension), are placed in one or more MQRFH2 headers in the message data. If a message handle is supplied then the behavior is to return the properties in the message handle.

MQPROP_COMPATIBILITY

If the message contains a property with a prefix of mcd., jms., usr. or mqext., all message properties are delivered to the application in an MQRFH2 header. Otherwise all properties of the message, except those contained in the message descriptor (or extension), are discarded and are no longer accessible to the application. This is the default value; it allows applications which expect JMS related properties to be in an MQRFH2 header in the message data to continue to work unmodified. If a message handle is supplied then the behavior is to return the properties in the message handle..

MQPROP_FORCE_MQRFH2

Properties are always returned in the message data in an MQRFH2 header regardless of whether the application specifies a message handle. A valid message handle supplied in the MsgHandle field of the MQGMO structure on the MQGET call is ignored. Properties of the message are not accessible via the message handle.

MQPROP_NONE

All properties of the message, except those in the message descriptor (or extension), are removed from the message before the message is delivered to the application. If a message handle is supplied then the behavior is to return the properties in the message handle.

This parameter is applicable to Local, Alias and Model queues. To determine its value, use the MQIA_PROPERTY_CONTROL selector with the MQINQ call.

QDepthHighEvent (MQLONG)

This controls whether Queue Depth High events are generated.

Local	Model	Alias	Remote	Cluster
X	X			

A Queue Depth High event indicates that an application has put a message on a queue, and this has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold (see the *QDepthHighLimit* attribute).

Note: The value of this attribute can change dynamically.

The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_Q_DEPTH_HIGH_EVENT selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QDepthHighLimit (MQLONG)

This is the threshold against which the queue depth is compared to generate a Queue Depth High event.

Local	Model	Alias	Remote	Cluster
X	X			

This event indicates that an application has put a message on a queue, and that this has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold. See [QDepthHighEvent](#) attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is greater than or equal to 0 and less than or equal to 100. The default value is 80.

To determine the value of this attribute, use the MQIA_Q_DEPTH_HIGH_LIMIT selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QDepthLowEvent (MQLONG)

This controls whether Queue Depth Low events are generated.

Local	Model	Alias	Remote	Cluster
X	X			

A Queue Depth Low event indicates that an application has retrieved a message from a queue, and that this has caused the number of messages on the queue to become less than or equal to the queue depth low threshold (see [QDepthLowLimit](#) attribute).

Note: The value of this attribute can change dynamically.

The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_Q_DEPTH_LOW_EVENT selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QDepthLowLimit (MQLONG)

This is the threshold against which the queue depth is compared to generate a Queue Depth Low event.

Local	Model	Alias	Remote	Cluster
X	X			

This event indicates that an application has retrieved a message from a queue, and that this has caused the number of messages on the queue to become less than or equal to the queue depth low threshold. See [QDepthLowEvent](#) attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is greater than or equal to 0 and less than or equal to 100. The default value is 20.

To determine the value of this attribute, use the MQIA_Q_DEPTH_LOW_LIMIT selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QDepthMaxEvent (MQLONG)

This controls whether Queue Full events are generated. A Queue Full event indicates that a put to a queue has been rejected because the queue is full, that is, the queue depth has already reached its maximum value.

Local	Model	Alias	Remote	Cluster
X	X			

Note: The value of this attribute can change dynamically.

The value is one of the following:

MQEVR_DISABLED

Event reporting disabled.

MQEVR_ENABLED

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_Q_DEPTH_MAX_EVENT selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QDesc (MQCHAR64)

Use this field for descriptive commentary.

Local	Model	Alias	Remote	Cluster
X	X	X	X	X

The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_Q_DESC selector with the MQINQ call. The length of this attribute is given by MQ_Q_DESC_LENGTH.

QName (MQCHAR48)

This is the name of a queue defined on the local queue manager.

Local	Model	Alias	Remote	Cluster
X		X	X	X

All queues defined on a queue manager share the same queue namespace. Therefore, an MQQT_LOCAL queue and an MQQT_ALIAS queue cannot have the same name.

To determine the value of this attribute, use the MQCA_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

QServiceInterval (MQLONG)

This is the service interval used for comparison to generate Service Interval High and Service Interval OK events.

Local	Model	Alias	Remote	Cluster
X	X			

See [QServiceIntervalEvent](#) attribute.

The value is in units of milliseconds, and is greater than or equal to zero, and less than or equal to 999 999 999.

To determine the value of this attribute, use the MQIA_Q_SERVICE_INTERVAL selector with the MQINQ call.

This attribute is supported on z/OS, but the MQINQ call cannot be used to determine its value.

QServiceIntervalEvent (MQLONG)

This controls whether Service Interval High or Service Interval OK events are generated.

Local	Model	Alias	Remote	Cluster
X	X			

- A Service Interval High event is generated when a check indicates that no messages have been retrieved from the queue for at least the time indicated by the *QServiceInterval* attribute.
- A Service Interval OK event is generated when a check indicates that messages have been retrieved from the queue within the time indicated by the *QServiceInterval* attribute.

Note: The value of this attribute can change dynamically.

The value is one of the following:

MQQSIE_HIGH

Queue Service Interval High events enabled.

- Queue Service Interval High events are **enabled** and
- Queue Service Interval OK events are **disabled**.

MQQSIE_OK

Queue Service Interval OK events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are **enabled**.

MQQSIE_NONE

No queue service interval events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are also **disabled**.

For shared queues, the value of this attribute is ignored; the value MQQSIE_NONE is assumed.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the MQIA_Q_SERVICE_INTERVAL_EVENT selector with the MQINQ call.

On z/OS, you cannot use the MQINQ call to determine the value of this attribute.

QSGDisp (MQLONG)

This specifies the disposition of the queue.

Local	Model	Alias	Remote	Cluster
X		X	X	

The value is one of the following:

MQQSGD_Q_MGR

The object has queue-manager disposition. This means that the object definition is known only to the local queue manager; the definition is not known to other queue managers in the queue-sharing group.

Each queue manager in the queue-sharing group can have an object with the same name and type as the current object, but these are separate objects and there is no correlation between them. Their attributes are not constrained to be the same as each other.

MQQSGD_COPY

The object is a local copy of a master object definition that exists in the shared repository. Each queue manager in the queue-sharing group can have its own copy of the object. Initially, all copies have the same attributes, but by using MQSC commands, you can alter each copy so that its attributes differ from those of the other copies. The attributes of the copies are resynchronized when the master definition in the shared repository is altered.

MQQSGD_SHARED

The object has shared disposition. This means that there exists in the shared repository a single instance of the object that is known to all queue managers in the queue-sharing group. When a queue manager in the group accesses the object, it accesses the single shared instance of the object.

To determine the value of this attribute, use the MQIA_QSG_DISP selector with the MQINQ call.

This attribute is supported only on z/OS.

QueueAccounting (MQLONG)

Local	Model	Alias	Remote	Cluster
X	X	X	X	

This controls the collection of accounting data for the queue. For accounting data to be collected for this queue, accounting data for this connection must also be enabled, using either the QMGR attribute ACCTQ or the Options field in the MQCNO structure on the MQCONN call.

This attribute has one of the following values:

MQMON_Q_MGR

Accounting data for this queue is collected based on the setting of the QMGR attribute ACCTQ. This is the default setting.

MQMON_OFF

Do not collect accounting data for this queue.

MQMON_ON

Collect accounting data for this queue.

To determine the value of this attribute, use the MQIA_ACCOUNTING_Q selector with the MQINQ call.

QueueMonitoring (MQLONG)

Controls the collection of online monitoring data for queues.

Local	Model	Alias	Remote	Cluster
X	X			

The value is one of the following:

MQMON_Q_MGR

Collect monitoring data according to the setting of the *QueueMonitoring* queue manager attribute. This is the default value.

MQMON_OFF

Online monitoring data collection is turned off for this queue.

MQMON_LOW

If the value of the *QueueMonitoring* queue manager attribute is not MQMON_NONE, online monitoring data collection is turned on, with a low rate of data collection for this queue.

MQMON_MEDIUM

If the value of the *QueueMonitoring* queue manager attribute is not MQMON_NONE, online monitoring data collection is turned on, with a moderate rate of data collection for this queue.

MQMON_HIGH

If the value of the *QueueMonitoring* queue manager attribute is not MQMON_NONE, online monitoring data collection is turned on, with a high rate of data collection for this queue.

To determine the value of this attribute, use the MQIA_MONITORING_Q selector with the MQINQ call.

QueueStatistics (MQCHAR12)

Local	Model	Alias	Remote	Cluster
X	X	X	X	

This controls the collection of statistics data for the queue.

This attribute has one of the following values:

MQMON_Q_MGR

Accounting data for this queue is collected based on the setting of the QMGR attribute STATQ. This is the default setting.

MQMON_OFF

Switch off statistics data collection for this queue.

MQMON_ON

Switch on statistics data collection for this queue.

QType (MQLONG)

Local	Model	Alias	Remote	Cluster
X		X	X	X

This is the type of queue; it has one of the following values:

MQQT_ALIAS

Alias queue definition.

MQQT_CLUSTER

Cluster queue.

MQQT_LOCAL

Local queue.

MQQT_REMOTE

Local definition of a remote queue.

To determine the value of this attribute, use the MQIA_Q_TYPE selector with the MQINQ call.

RemoteQMgrName (MQCHAR48)

Local	Model	Alias	Remote	Cluster
			X	

This is the name of the remote queue manager on which the queue *RemoteQName* is defined. If the *RemoteQName* queue has a *QSGDisp* value of MQQSGD_COPY or MQQSGD_SHARED, *RemoteQMgrName* can be the name of the queue-sharing group that owns *RemoteQName*.

If an application opens the local definition of a remote queue, *RemoteQMgrName* must not be blank and must not be the name of the local queue manager. If *XmitQName* is blank, the local queue with the same name as *RemoteQMgrName* is used as the transmission queue. If there is no queue with the name *RemoteQMgrName*, the queue identified by the *DefXmitQName* queue-manager attribute is used.

If this definition is used for a queue-manager alias, *RemoteQMgrName* is the name of the queue manager that is being aliased. It can be the name of the local queue manager. Otherwise, if *XmitQName* is blank when the open occurs, there must be a local queue with a name that is the same as *RemoteQMgrName* ; this queue is used as the transmission queue.

If this definition is used for a reply-to alias, this name is the name of the queue manager that is to be the *ReplyToQMgr*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the MQCA_REMOTE_Q_MGR_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_NAME_LENGTH.

RemoteQName (MQCHAR48)

Local	Model	Alias	Remote	Cluster
			X	

This is the name of the queue as it is known on the remote queue manager *RemoteQMgrName*.

If an application opens the local definition of a remote queue, when the open occurs *RemoteQName* must not be blank.

If this definition is used for a queue-manager alias definition, when the open occurs *RemoteQName* must be blank.

If the definition is used for a reply-to alias, this name is the name of the queue that is to be the *ReplyToQ*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the MQCA_REMOTE_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

RetentionInterval (MQLONG)

This is the period of time for which to retain the queue. After this time has elapsed, the queue is eligible for deletion.

Local	Model	Alias	Remote	Cluster
X	X			

The time is measured in hours, counting from the date and time when the queue was created. The creation date and time of the queue are recorded in the *CreationDate* and *CreationTime* attributes.

This information is provided to enable a housekeeping application or the operator to identify and delete queues that are no longer required.

Note: The queue manager never takes any action to delete queues based on this attribute, or to prevent the deletion of queues with a retention interval that has not expired; it is the user's responsibility to take any required action.

Use a realistic retention interval to prevent the accumulation of permanent dynamic queues (see [DefinitionType attribute](#)). However, this attribute can also be used with predefined queues.

To determine the value of this attribute, use the MQIA_RETENTION_INTERVAL selector with the MQINQ call.

Scope (MQLONG)

This controls whether an entry for this queue also exists in a cell directory.

Local	Model	Alias	Remote	Cluster
X		X	X	

A cell directory is provided by an installable Name service. The value is one of the following:

MQSCO_Q_MGR

The queue definition has queue-manager scope: the definition of the queue does not extend beyond the queue manager that owns it. To open the queue for output from some other queue manager, either the name of the owning queue manager must be specified, or the other queue manager must have a local definition of the queue.

MQSCO_CELL

The queue definition has cell scope: the queue definition is also placed in a cell directory available to all the queue managers in the cell. The queue can be opened for output from any of the queue managers in the cell by specifying the name of the queue; the name of the queue manager that owns the queue need not be specified. However, the queue definition is not available to any queue manager in the cell that also has a local definition of a queue with that name, as the local definition takes precedence.

A cell directory is provided by an installable Name service.

Model and dynamic queues cannot have cell scope.

This value is only valid if a name service supporting a cell directory has been configured.

To determine the value of this attribute, use the MQIA_SCOPE selector with the MQINQ call.

Support for this attribute is subject to the following restrictions:

- On IBM i, the attribute is supported, but only MQSCO_Q_MGR is valid.
- On z/OS, the attribute is not supported.

Shareability (MQLONG)

This indicates whether the queue can be opened for input multiple times concurrently.

Local	Model	Alias	Remote	Cluster
X	X			

The value is one of the following:

MQQA_SHAREABLE

Queue is shareable.

Multiple opens with the MQOO_INPUT_SHARED option are allowed.

MQQA_NOT_SHAREABLE

Queue is not shareable.

An MQOPEN call with the MQOO_INPUT_SHARED option is treated as MQOO_INPUT_EXCLUSIVE.

To determine the value of this attribute, use the MQIA_SHAREABILITY selector with the MQINQ call.

StorageClass (MQCHAR8)

This is a user-defined name that defines the physical storage used to hold the queue. In practice, a message is written to disk only if it needs to be paged out of its memory buffer.

Local	Model	Alias	Remote	Cluster
X	X			

To determine the value of this attribute, use the MQCA_STORAGE_CLASS selector with the MQINQ call. The length of this attribute is given by MQ_STORAGE_CLASS_LENGTH.

This attribute is supported only on z/OS.

TriggerControl (MQLONG)

This controls whether trigger messages are written to an initiation queue to start an application to service the queue.

Local	Model	Alias	Remote	Cluster
X	X			

This is one of the following:

MQTC_OFF

No trigger messages are to be written for this queue. The value of *TriggerType* is irrelevant in this case.

MQTC_ON

Trigger messages are to be written for this queue when the appropriate trigger events occur.

To determine the value of this attribute, use the MQIA_TRIGGER_CONTROL selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerData (MQCHAR64)

This is free-format data that the queue manager inserts into the trigger message when a message arriving on this queue causes a trigger message to be written to the initiation queue.

Local	Model	Alias	Remote	Cluster
X	X			

The content of this data is of no significance to the queue manager. It is meaningful either to the trigger-monitor application that processes the initiation queue, or to the application that the trigger monitor starts.

The character string must not contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_TRIGGER_DATA selector with the MQINQ call. To change the value of this attribute, use the MQSET call. The length of this attribute is given by MQ_TRIGGER_DATA_LENGTH.

TriggerDepth (MQLONG)

Local	Model	Alias	Remote	Cluster
X	X			

This is the number of messages of priority *TriggerMsgPriority* or greater that must be on the queue before a trigger message is written. This applies when *TriggerType* is set to MQTT_DEPTH. The value of *TriggerDepth* is one or greater. This attribute is not used otherwise.

To determine the value of this attribute, use the MQIA_TRIGGER_DEPTH selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerMsgPriority (MQLONG)

This is the message priority below which messages do not contribute to the generation of trigger messages (that is, the queue manager ignores these messages when deciding whether to generate a trigger message).

Local	Model	Alias	Remote	Cluster
X	X			

TriggerMsgPriority can be in the range zero (lowest) through *MaxPriority* (highest; see [MaxPriority attribute](#)); a value of zero causes all messages to contribute to the generation of trigger messages.

To determine the value of this attribute, use the MQIA_TRIGGER_MSG_PRIORITY selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerType (MQLONG)

This controls the conditions under which trigger messages are written as a result of messages arriving on this queue.

Local	Model	Alias	Remote	Cluster
X	X			

It has one of the following values:

MQTT_NONE

No trigger messages are written as a result of messages on this queue. This has the same effect as setting *TriggerControl* to MQTC_OFF.

MQTT_FIRST

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue changes from 0 to 1.

MQTT EVERY

A trigger message is written whenever a message of priority *TriggerMsgPriority* or greater arrives on the queue.

MQTT_DEPTH

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue equals or exceeds *TriggerDepth*. After the trigger message has been written,

TriggerControl is set to MQTC_OFF to prevent further triggering until it is explicitly turned on again.

To determine the value of this attribute, use the MQIA_TRIGGER_TYPE selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

Usage (MQLONG)

This indicates what the queue is used for.

Local	Model	Alias	Remote	Cluster
X	X			

The value is one of the following:

MQUS_NORMAL

This is a queue that applications use when putting and getting messages; the queue is not a transmission queue.

MQUS_TRANSMISSION

This is a queue used to hold messages destined for remote queue managers. When an application sends a message to a remote queue, the local queue manager stores the message temporarily on the appropriate transmission queue in a special format. A message channel agent then reads the message from the transmission queue, and transports the message to the remote queue manager. For more information about transmission queues, see [Defining a transmission queue](#).

Only privileged applications can open a transmission queue for MQOO_OUTPUT to put messages on it directly. Usually, only utility applications do this. Ensure that the message data format is correct (see “MQXQH - Transmission-queue header” on page 592) or errors might occur during the transmission process. Context is not passed or set unless one of the MQPMO_*_CONTEXT context options is specified.

To determine the value of this attribute, use the MQIA_USAGE selector with the MQINQ call.

XmitQName (MQCHAR48)

This is the transmission queue name. If this attribute is nonblank when an open occurs, either for a remote queue or for a queue-manager alias definition, it specifies the name of the local transmission queue to be used for forwarding the message.

Local	Model	Alias	Remote	Cluster
			X	

If *XmitQName* is blank, the local queue with a name that is the same as *RemoteQMGrName* is used as the transmission queue. If there is no queue with the name *RemoteQMGrName*, the queue identified by the *DefXmitQName* queue-manager attribute is used.

This attribute is ignored if the definition is being used as a queue-manager alias and *RemoteQMGrName* is the name of the local queue manager. It is also ignored if the definition is used as a reply-to queue alias definition.

To determine the value of this attribute, use the MQCA_XMIT_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

Attributes for namelists

The following table summarizes the attributes that are specific to namelists. The attributes are described in alphabetical order.

Namelists are supported on all IBM MQ systems, plus IBM MQ MQI clients connected to these systems.

Note: The names of the attributes shown in this section are descriptive names used with the MQINQ and MQSET calls; the names are the same as for the PCF commands. When MQSC commands are used to

define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) Commands](#) for more information.

Table 564. Attributes for namelists	
Attribute	Description
AlterationDate	Date when definition was last changed
AlterationTime	Time when definition was last changed
NameCount	Number of names in namelist
NamelistDesc	Namelist description
NamelistName	Namelist name
Names	A list of <i>NameCount</i> names
NamelistType	Namelist type
QSGDisp	Queue-sharing group disposition

AlterationDate (MQCHAR12)

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the MQCA_ALTERATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_DATE_LENGTH.

AlterationTime (MQCHAR8)

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the MQCA_ALTERATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_TIME_LENGTH.

NameCount (MQLONG)

This is the number of names in the namelist. It is greater than or equal to zero. The following value is defined:

MQNC_MAX_NAMELIST_NAME_COUNT

Maximum number of names in a namelist.

To determine the value of this attribute, use the MQIA_NAME_COUNT selector with the MQINQ call.

NamelistDesc (MQCHAR64)

Use this field for descriptive commentary; its value is established by the definition process. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_NAMELIST_DESC selector with the MQINQ call.

The length of this attribute is given by MQ_NAMELIST_DESC_LENGTH.

NamelistName (MQCHAR48)

This is the name of a namelist that is defined on the local queue manager. For more information about namelist names, see the [Other object names](#) section.

Each namelist has a name that is different from the names of other namelists belonging to the queue manager, but might duplicate the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the MQCA_NAMELIST_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_NAMELIST_NAME_LENGTH.

NamelistType (MQLONG)

This specifies the nature of the names in the namelist, and indicates how the namelist is used. It is one of the following values:

MQNT_NONE

Namelist with no assigned type.

MQNT_Q

Namelist containing the names of queues.

MQNT_CLUSTER

Namelist containing the names of clusters.

MQNT_AUTH_INFO

Namelist containing the names of authentication-information objects.

To determine the value of this attribute, use the MQIA_NAMELIST_TYPE selector with the MQINQ call.

This attribute is supported only on z/OS.

Names (MQCHAR48xNameCount)

This is a list of *NameCount* names, where each name is the name of an object that is defined to the local queue manager. For more information about object names, see [Rules for naming IBM MQ objects](#).

To determine the value of this attribute, use the MQCA_NAMES selector with the MQINQ call.

The length of each name in the list is given by MQ_OBJECT_NAME_LENGTH.

QSGDisp (MQLONG)

This specifies the disposition of the namelist. The value is one of the following:

MQQSGD_Q_MGR

The object has queue-manager disposition: the object definition is known only to the local queue manager; the definition is not known to other queue managers in the queue-sharing group.

Each queue manager in the queue-sharing group can have an object with the same name and type as the current object, but these are separate objects and there is no correlation between them. Their attributes are not constrained to be the same as each other.

MQQSGD_COPY

The object is a local copy of a master object definition that exists in the shared repository. Each queue manager in the queue-sharing group can have its own copy of the object. Initially, all copies have the same attributes, but you can alter each copy, using MQSC commands, so that its attributes differ from those of the other copies. The attributes of the copies are resynchronized when the master definition in the shared repository is altered.

To determine the value of this attribute, use the MQIA_QSG_DISP selector with the MQINQ call.

This attribute is supported only on z/OS.

Attributes for process definitions

The following table summarizes the attributes that are specific to process definitions. The attributes are described in alphabetical order.

Note: The names of the attributes in this section are descriptive names used with the MQINQ and MQSET calls; the names are the same as for the PCF commands. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) Commands](#) for more information.

Table 565. Attributes for process definitions	
Attribute	Description
AlterationDate	Date when definition was last changed
AlterationTime	Time when definition was last changed
ApplId	Application identifier
ApplType	Application type
EnvData	Environment data
ProcessDesc	Process description
ProcessName	Process name
QSGDisp	Queue-sharing group disposition
UserData	User data

AlterationDate (MQCHAR12)

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the MQCA_ALTERATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_DATE_LENGTH.

AlterationTime (MQCHAR8)

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the MQCA_ALTERATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_TIME_LENGTH.

ApplId (MQCHAR256)

This is a character string that identifies the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

The meaning of *ApplId* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ requires *ApplId* to be the name of an executable program. The following notes apply to the environments indicated:

- On z/OS, *ApplId* must be:
 - A CICS transaction identifier, for applications started using the CICS trigger-monitor transaction CKTI
 - An IMS transaction identifier, for applications started using the IMS trigger monitor CSQQTRMN
- On Windows systems, the program name can be prefixed with a drive and directory path.
- On UNIX systems, the program name can be prefixed with a directory path.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_APPL_ID selector with the MQINQ call. The length of this attribute is given by MQ_PROCESS_APPL_ID_LENGTH.

ApplType (MQLONG)

Application types identify the nature of the program to be started in response to the receipt of a trigger message. This information is used by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

ApplType can have any value, but the following values are recommended for standard types; restrict user-defined application types to values in the range MQAT_USER_FIRST through MQAT_USER_LAST:

MQAT_AIX

AIX application (same value as MQAT_UNIX).

MQAT_BATCH

Batch application

MQAT_BROKER

Broker application

MQAT_CICS

CICS transaction.

MQAT_CICS_BRIDGE

CICS bridge application.

MQAT_CICS_VSE

CICS/VSE transaction.

MQAT_DOS

IBM MQ MQI client application on PC DOS.

MQAT_IMS

IMS application.

MQAT_IMS_BRIDGE

IMS bridge application.

MQAT_JAVA

Java application.

MQAT_MVS

MVS or TSO application (same value as MQAT_ZOS).

MQAT_NOTES_AGENT

Lotus Notes Agent application.

MQAT_NSK

HP Integrity NonStop Server application.

MQAT_OS390

OS/390 application (same value as MQAT_ZOS).

MQAT_OS400

IBM i application.

MQAT_RRS_BATCH

RRS batch application.

MQAT_UNIX

UNIX application.

MQAT_UNKNOWN

Application of unknown type.

MQAT_USER

User application.

MQAT_VOS

Stratus VOS application.

MQAT_WINDOWS

16-bit Windows application.

MQAT_WINDOWS_NT

32-bit Windows application.

MQAT_WLM

z/OS workload manager application.

MQAT_XCF

XCF.

MQAT_ZOS

z/OS application.

MQAT_USER_FIRST

Lowest value for user-defined application type.

MQAT_USER_LAST

Highest value for user-defined application type.

To determine the value of this attribute, use the MQIA_APPL_TYPE selector with the MQINQ call.

EnvData (MQCHAR128)

This is a character string that contains environment-related information pertaining to the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

The meaning of *EnvData* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ appends *EnvData* to the parameter list passed to the started application. The parameter list consists of the MQTMC2 structure, followed by one blank, followed by *EnvData* with trailing blanks removed. The following notes apply to the environments indicated:

- On z/OS:
 - *EnvData* is not used by the trigger-monitor applications provided by IBM MQ.
 - If ApplType is MQAT_WLM, you can supply default values in *EnvData* for the ServiceName and ServiceStep fields in the work information header (MQWIH).
- On UNIX systems, *EnvData* can be set to the & character to run the started application in the background.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_ENV_DATA selector with the MQINQ call. The length of this attribute is given by MQ_PROCESS_ENV_DATA_LENGTH.

ProcessDesc (MQCHAR64)

Use this field for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_PROCESS_DESC selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_DESC_LENGTH.

ProcessName (MQCHAR48)

This is the name of a process definition that is defined on the local queue manager.

Each process definition has a name that is different from the names of other process definitions belonging to the queue manager. But the name of the process definition might be the same as the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the MQCA_PROCESS_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_NAME_LENGTH.

QSGDisp (MQLONG)

This specifies the disposition of the process definition. The value is one of the following:

MQQSGD_Q_MGR

The object has queue-manager disposition: the object definition is known only to the local queue manager; the definition is not known to other queue managers in the queue-sharing group.

Each queue manager in the queue-sharing group can have an object with the same name and type as the current object, but these are separate objects and there is no correlation between them. Their attributes are not constrained to be the same as each other.

MQQSGD_COPY

The object is a local copy of a master object definition that exists in the shared repository. Each queue manager in the queue-sharing group can have its own copy of the object. Initially, all copies have the same attributes, but you can alter each copy, using MQSC commands, so that its attributes differ from those of the other copies. The attributes of the copies are resynchronized when the master definition in the shared repository is altered.

To determine the value of this attribute, use the MQIA_QSG_DISP selector with the MQINQ call.

This attribute is supported only on z/OS.

UserData (MQCHAR128)

UserData is a character string that contains user information pertaining to the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue, or the application that is started by the trigger monitor. The information is sent to the initiation queue as part of the trigger message.

The meaning of *UserData* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ passes *UserData* to the started application as part of the parameter list. The parameter list consists of the MQTMC2 structure (containing *UserData*), followed by one blank, followed by *EnvData* with trailing blanks removed.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary. For Microsoft Windows, the character string must not contain double quotation marks if the process definition is going to be passed to **runmqtrm**.

To determine the value of this attribute, use the MQCA_USER_DATA selector with the MQINQ call. The length of this attribute is given by MQ_PROCESS_USER_DATA_LENGTH.

Return codes

For each IBM MQ Message Queue Interface (MQI) and IBM MQ Administration Interface (MQAI) call, a **completion** code and a **reason** code are returned by the queue manager or by an exit routine, to indicate the success or failure of the call.

Applications must not depend upon errors being checked for in a specific order, except where specifically noted. If more than one completion code or reason code could arise from a call, the particular error reported depends on the implementation.

Applications checking for successful completion following an IBM MQ API call must always check the completion code. Do not assume the completion code value, based on the value of the reason code.

Completion codes

The completion code parameter (*CompCode*) allows the caller to see quickly whether the call completed successfully, completed partially, or failed. The following is a list of completion codes, with more detail than is given in the call descriptions:

MQCC_OK

The call completed fully; all output parameters have been set. The *Reason* parameter always has the value MQRC_NONE in this case.

MQCC_WARNING

The call completed partially. Some output parameters might have been set in addition to the *CompCode* and *Reason* output parameters. The *Reason* parameter gives additional information about the partial completion.

MQCC_FAILED

The processing of the call did not complete. The state of the queue manager is unchanged, except where specifically noted. The *CompCode* and *Reason* output parameters have been set; other parameters are unchanged, except where noted.

The reason might be a fault in the application program, or it might be the result of some situation external to the program, for example the user's authority might have been revoked. The *Reason* parameter gives additional information about the error.

Reason codes

The reason code parameter (*Reason*) qualifies the completion code parameter (*CompCode*).

If there is no special reason to report, MQRC_NONE is returned. A successful call returns MQCC_OK and MQRC_NONE.

If the completion code is either MQCC_WARNING or MQCC_FAILED, the queue manager always reports a qualifying reason; details are given under each call description.

Where user exit routines set completion codes and reasons, they must adhere to these rules. In addition, any special reason values defined by user exits must be less than zero, to ensure that they do not conflict with values defined by the queue manager. Exits can set reasons already defined by the queue manager, where appropriate.

Reason codes also occur in:

- The *Reason* field of the MQDLH structure
- The *Feedback* field of the MQMD structure

For complete descriptions of reason codes, see [Messages and reason codes](#).

Rules for validating MQI options

This section lists the situations that produce an MQRC_OPTIONS_ERROR reason code from an MQOPEN, MQPUT, MQPUT1, MQGET, MQCLOSE, or MQSUB call.

MQOPEN call

For the options of the MQOPEN call:

- At least *one* of the following must be specified:
 - MQOO_BROWSE
 - MQOO_INPUT_EXCLUSIVE ¹
 - MQOO_INPUT_SHARED ¹
 - MQOO_INPUT_AS_Q_DEF ¹
 - MQOO_INQUIRE
 - MQOO_OUTPUT
 - MQOO_SET
 - MQOO_BIND_ON_OPEN ²
 - MQOO_BIND_NOT_FIXED ²
 - MQOO_BIND_ON_GROUP ²
 - MQOO_BIND_AS_Q_DEF ²
- Only *one* of the following is allowed:

- MQOO_READ_AHEAD
- MQOO_NO_READ_AHEAD
- MQOO_READ_AHEAD_AS_Q_DEF

1. Only *one* of the following is allowed:

- MQOO_INPUT_EXCLUSIVE
- MQOO_INPUT_SHARED
- MQOO_INPUT_AS_Q_DEF

2. Only *one* of the following is allowed:

- MQOO_BIND_ON_OPEN
- MQOO_BIND_NOT_FIXED
- MQOO_BIND_ON_GROUP
- MQOO_BIND_AS_Q_DEF

Note: The options that are listed previously are mutually exclusive. However, as the value of MQOO_BIND_AS_Q_DEF is zero, specifying it with either of the other two bind options does not result in reason code MQRC_OPTIONS_ERROR. MQOO_BIND_AS_Q_DEF is provided to aid program documentation.

- If MQOO_SAVE_ALL_CONTEXT is specified, one of the MQOO_INPUT_* options must also be specified.
- If one of the MQOO_SET_*_CONTEXT or MQOO_PASS_*_CONTEXT options are specified, MQOO_OUTPUT must also be specified.
- If MQOO_CO_OP is specified, MQOO_BROWSE must also be specified
- If MQOO_NO_MULTICAST is specified, MQOO_OUTPUT must also be specified.

MQPUT call

For the put-message options:

- The combination of MQPMO_SYNCPOINT and MQPMO_NO_SYNCPOINT is not allowed.
- Only *one* of the following is allowed:
 - MQPMO_DEFAULT_CONTEXT
 - MQPMO_NO_CONTEXT
 - MQPMO_PASS_ALL_CONTEXT
 - MQPMO_PASS_IDENTITY_CONTEXT
 - MQPMO_SET_ALL_CONTEXT
 - MQPMO_SET_IDENTITY_CONTEXT
- Only *one* of the following is allowed:
 - MQPMO_ASYNC_RESPONSE
 - MQPMO_SYNC_RESPONSE
 - MQPMO_RESPONSE_AS_TOPIC_DEF
 - MQPMO_RESPONSE_AS_Q_DEF
- MQPMO_ALTERNATE_USER_AUTHORITY is not allowed (it is valid only on the MQPUT1 call).

MQPUT1 call

For the put-message options, the rules are the same as for the MQPUT call, except for the following:

- MQPMO_ALTERNATE_USER_AUTHORITY is allowed.
- MQPMO_LOGICAL_ORDER is *not* allowed.

MQGET call

For the get-message options:

- Only *one* of the following is allowed:
 - MQGMO_NO_SYNCPOINT
 - MQGMO_SYNCPOINT
 - MQGMO_SYNCPOINT_IF_PERSISTENT
- Only *one* of the following is allowed:
 - MQGMO_BROWSE_FIRST
 - MQGMO_BROWSE_MSG_UNDER_CURSOR
 - MQGMO_BROWSE_NEXT
 - MQGMO_MSG_UNDER_CURSOR
- MQGMO_SYNCPOINT is not allowed with any of the following:
 - MQGMO_BROWSE_FIRST
 - MQGMO_BROWSE_MSG_UNDER_CURSOR
 - MQGMO_BROWSE_NEXT
 - MQGMO_LOCK
 - MQGMO_UNLOCK
- MQGMO_SYNCPOINT_IF_PERSISTENT is not allowed with any of the following:
 - MQGMO_BROWSE_FIRST
 - MQGMO_BROWSE_MSG_UNDER_CURSOR
 - MQGMO_BROWSE_NEXT
 - MQGMO_COMPLETE_MSG
 - MQGMO_UNLOCK
- MQGMO_MARK_SKIP_BACKOUT requires MQGMO_SYNCPOINT to be specified.
- The combination of MQGMO_WAIT and MQGMO_SET_SIGNAL is not allowed.
- If MQGMO_LOCK is specified, one of the following must also be specified:
 - MQGMO_BROWSE_FIRST
 - MQGMO_BROWSE_MSG_UNDER_CURSOR
 - MQGMO_BROWSE_NEXT
- If MQGMO_UNLOCK is specified, only the following are allowed:
 - MQGMO_NO_SYNCPOINT
 - MQGMO_NO_WAIT

MQCLOSE call

For the options of the MQCLOSE call:

- The combination of MQCO_DELETE and MQCO_DELETE_PURGE is not allowed.
- Only one of the following is allowed:
 - MQCO_KEEP_SUB
 - MQCO_REMOVE_SUB

MQSUB call

For the options of the MQSUB call:

- At least one of the following must be specified:

- MQSO_ALTER
- MQSO_RESUME
- MQSO_CREATE

- Only one of the following is allowed:

- MQSO_DURABLE
- MQSO_NON_DURABLE

Note: The options that are listed previously are mutually exclusive. However, as the value of MQSO_NON_DURABLE is zero, specifying it with MQSO_DURABLE does not result in reason code MQRC_OPTIONS_ERROR. MQSO_NON_DURABLE is provided to aid program documentation.

- The combination of MQSO_GROUP_SUB and MQSO_MANAGED is not allowed.
- MQSO_GROUP_SUB requires MQSO_SET_CORREL_ID to be specified.
- Only one of the following is allowed:
 - MQSO_ANY_USERID
 - MQSO_FIXED_USERID
- MQSO_NEW_PUBLICATIONS_ONLY is allowed in combination with:
 - MQSO_CREATE
 - MQSO_ALTER, if MQSO_NEW_PUBLICATIONS_ONLY was set on the original subscription
- The combination of MQSO_PUBLICATIONS_ON_REQUEST and SubLevel greater than 1 is not allowed.
- Only one of the following is allowed:
 - MQSO_WILDCARD_CHAR
 - MQSO_WILDCARD_TOPIC
- MQSO_NO_MULTICAST requires MQSO_MANAGED to be specified.

Queued publish/subscribe command messages

An application can use MQRFH2 command messages to control a queued publish/subscribe application.

An application that is using MQRFH2 for publish/subscribe can send the following command messages to the SYSTEM.BROKER.CONTROL.QUEUE:

- [“Delete Publication message” on page 843](#)
- [“Deregister Subscriber message” on page 844](#)
- [“Publish message” on page 847](#)
- [“Register Subscriber message” on page 850](#)
- [“Request Update message” on page 854](#)

If you are writing queued publish/subscribe applications, you must understand these messages, the queue manager response message, and the message descriptor (MQMD); see the following information:

- [“Queue Manager Response message” on page 856](#)
- [“MQMD settings for publications forwarded by a queue manager” on page 861](#)
- [“MQMD settings in queue manager response messages” on page 862](#)
- [“Publish/subscribe reason codes” on page 858](#)

The commands are contained in a <psc> folder in the **NameValueData** field of the MQRFH2 header. The message that can be sent by a broker in response to a command message is contained in a <pscr> folder.

The descriptions of each command list the properties that can be contained in a folder. Unless otherwise specified, the properties are optional and can occur only once.

Names of properties are shown as `<Command>`.

Values must be in string format, for example: `Publish`.

A string constant representing the value of a property is shown in parentheses, for example: `(MQPSC_PUBLISH)`.

String constants are defined in the header file `cmqpsc.h` which is supplied with the queue manager.

Delete Publication message

The **Delete Publication** command message is sent to a queue manager from a publisher, or from another queue manager, to tell the queue manager to delete any retained publications for the specified topics.

This message is sent to a queue monitored by the queue manager's queued publish/subscribe interface.

The input queue should be the queue that the original publication was sent to.

If you have the authority for some, but not all, of the topics that are specified in the **Delete Publication** command message, only those topics are deleted. A **Broker Response** message indicates which topics are not deleted.

Similarly, if a **Publish** command contains more than one topic, a **Delete Publication** command matching some, but not all, of those topics deletes only the publications for the topics that are specified in the **Delete Publication** command.

See “MQMD settings for publications forwarded by a queue manager” on page 861 for details of the message descriptor (MQMD) parameters that are needed when sending a command message to the queue manager.

Properties

<Command> (MQPSC_COMMAND)

The value is `DeletePub (MQPSC_DELETE_PUBLICATION)`.

This property must be specified.

<Topic> (MQPSC_TOPIC)

The value is a string that contains a topic for which retained publications are to be deleted. Wildcard characters can be included in the string to delete publications on more than one topic.

This property must be specified; it can be repeated for as many topics as needed.

<DelOpt> (MQPSC_DELETE_OPTION)

The delete options property can take one of the following values:

Local (MQPSC_LOCAL)

All retained publications for the specified topics are deleted at the local queue manager (that is, the queue manager to which this message is sent), whether they were published with the `Local` option or not.

Publications at other queue managers are not affected.

None (MQPSC_NONE)

All options take their default values. This has the same effect as omitting the `DelOpt` property. If other options are specified at the same time, `None` is ignored.

The default if this property is omitted is that all retained publications for the specified topics are deleted at all queue managers in the network, regardless of whether they were published with the `Local` option.

Example

Here is an example of NameValueData for a **Delete Publication** command message. This is used by the sample application to delete, at the local queue manager, the retained publication that contains the latest score in the match between Team1 and Team2.

```
<psc>
  <Command>DeletePub</Command>
  <Topic>Sport/Soccer/State/LatestScore/Team1 Team2</Topic>
  <DelOpt>Local</DelOpt>
</psc>
```

Deregister Subscriber message

The **Deregister Subscriber** command message is sent to a queue manager by a subscriber, or by another application on behalf of a subscriber, to indicate that it no longer wants to receive messages matching the given parameters.

This message is sent to SYSTEM.BROKER.CONTROL.QUEUE, the queue manager's control queue. The user must have the necessary authority to put a message onto this queue.

See [MQMD settings for publications forwarded by a queue manager](#) for details of the message descriptor (MQMD) parameters that are needed when sending a command message to the queue manager.

An individual subscription can be deregistered by specifying the corresponding topic, subscription point and filter values of the original subscription. If any of the values were not specified (that is, they took the default values) in the original subscription, they should be omitted when the subscription is deregistered.

All subscriptions for a subscriber, or a group of subscribers, can be deregistered by using the DeregAll option. For example, if DeregAll is specified, together with a subscription point (but no topic or filter), then all subscriptions for the subscriber on the specified subscription point are deregistered, regardless of the topic and filter. Any combination of topic, filter and subscription point is allowed; if all three are specified only one subscription can match, and the DeregAll option is ignored.

The message must be sent by the subscriber that registered the subscription; this is confirmed by checking the subscriber's user ID.

Subscriptions can also be deregistered by a system administrator using MQSC or PCF commands. However, the subscriptions registered with a temporary dynamic queue are associated with the queue, not just the queue name. If the queue is deleted, either explicitly, or by the application disconnecting from the queue manager, it is no longer possible to use the **Deregister Subscriber** command to deregister the subscriptions for that queue. The subscriptions can be deregistered using the developer workbench, and they are removed automatically by the queue manager the next time that it matches a publication to the subscription, or the next time the queue manager restarts. Under normal circumstances, applications should deregister their subscriptions before deleting the queue, or disconnecting from the queue manager.

If a subscriber sends a message to deregister a subscription, and receives a response message to say that this was processed successfully, some publications might still reach the subscriber queue if they were being processed by the queue manager at the same time as the subscription was being deregistered. If the messages are not removed from the queue, there might be a buildup of unprocessed messages on the subscriber queue. If the application executes a loop that includes an MQGET call with the appropriate CorrelId after sleeping for a while, these messages are cleared off the queue.

Similarly, if the subscriber uses a permanent dynamic queue, and deregisters and closes the queue with the MQCO_DELETE_PURGE option on an MQCLOSE call, the queue might not be empty. If any publications from the queue manager are not yet committed when the queue is deleted, an MQRC_Q_NOT_EMPTY return code is issued by the MQCLOSE call. The application can avoid this problem by sleeping and reissuing the MQCLOSE call from time to time.

Properties

<Command> (MQPSC_COMMAND)

The value is DeregSub (MQPSC_DEREGISTER_SUBSCRIBER).

This property must be specified.

<Topic> (MQPSC_TOPIC)

The value is a string that contains the topic to be deregistered.

This property can, optionally, be repeated if multiple topics are to be deregistered. It can be omitted if DeregAll is specified in <RegOpt>.

The topics that are specified can be a subset of those that are registered if the subscriber wants to retain subscriptions for other topics. Wildcard characters are allowed, but a topic string that contains wildcard characters must exactly match the corresponding string that was specified in the **Deregister Subscriber** command message.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is a string that specifies the subscription point from which the subscription is to be detached.

This property must not be repeated. It can be omitted if a <Topic> is specified, or if DeregAll is specified in <RegOpt>. If you omit this property, the following happens:

- If you do **not** specify DeregAll, subscriptions matching the <Topic> property (and the <Filter> property, if present) are deregistered from the default subscription point.
- If you specify DeregAll, all subscriptions (matching the <Topic> and <Filter> properties if present) are deregistered from all subscription points.

Note that you cannot specify the default subscription point explicitly. Therefore, there is no way of deregistering all subscriptions from this subscription point only; you must specify the topics.

<SubIdentity> (MQPSC_SUBSCRIPTION_IDENTITY)

This is a variable-length string with a maximum length of 64 characters. It is used to represent an application with an interest in a subscription. The queue manager maintains a set of subscriber identities for each subscription. Each subscription can allow its identity set to hold only a single identity, or an unlimited number of identities.

If the SubIdentity is in the identity set for the subscription then it is removed from the set. If the identity set becomes empty as a result of this, the subscription is removed from the queue manager, unless LeaveOnly is specified as a value of the RegOpt property. If the identity set still contains other identities then the subscription is not removed from the queue manager, and publication flow is not interrupted.

If SubIdentity is specified, but the SubIdentity is not in the identity set for the subscription, then the **Deregister Subscriber** command fails with the return code *MQRCCF_SUB_IDENTITY_ERROR*.

<Filter> (MQPSC_FILTER)

The value is a string specifying the filter to be deregistered. It must match exactly, including case and any spaces, a subscription filter that has been previously registered.

This property can, optionally, be repeated if more than one filter is to be deregistered. It can be omitted if a <Topic> is specified, or if DeregAll is specified in <RegOpt>.

The filters specified can be a subset of those registered if the subscriber wants to retain subscriptions for other filters.

<RegOpt> (MQPSC_REGISTRATION_OPTION)

The registration options property can take the following values:

DeregAll

(MQPSC_DEREGISTER_ALL)

All matching subscriptions registered for this subscriber are to be deregistered.

If you specify DeregAll:

- <Topic>, <SubPoint>, and <Filter> can be omitted.
- <Topic> and <Filter> can be repeated, if required.
- <SubPoint> must not be repeated.

If you do **not** specify **DeregAll**:

- <Topic> must be specified, and can be repeated if required.
- <SubPoint> and <Filter> can be omitted.
- <SubPoint> must not be repeated.
- <Filter> can be repeated, if required.

If topics and filters are both repeated, then all subscriptions matching all combinations of the two are removed. For example, a **Deregister Subscriber** command that specifies three topics and three filters will attempt to remove nine subscriptions.

CorrelAsId

(MQPSC_CORREL_ID_AS_IDENTITY)

The CorrelId in the message descriptor (MQMD), which must not be zero, is used to identify the subscriber. It must match the CorrelId used in the original subscription.

FullResp

(MQPSC_FULL_RESPONSE)

When FullResp is specified all attributes of the subscription are returned in the response message, if the command does not fail.

When FullResp is specified DeregAll is not permitted in the **Deregister Subscriber** command. It is also not possible to specify multiple topics. The command fails with return code *MQRCCF_REG_OPTIONS_ERROR*, in both cases.

LeaveOnly

(MQPSC_LEAVE_ONLY)

When you specify this with a SubIdentity which is in the identity set for the subscription the SubIdentity is removed from the identity set for the subscription. The subscription is not removed from the queue manager, even if the resulting identity set is empty. If the SubIdentity value is not in the identity set the command fails with return code *MQRCCF_SUB_IDENTITY_ERROR*.

If LeaveOnly is specified with no SubIdentity, the command fails with return code *MQRCCF_REG_OPTIONS_ERROR*.

If neither LeaveOnly nor a SubIdentity are specified, then the subscription is removed regardless of the contents of the identity set for the subscription.

None

(MQPSC_NONE)

All options take their default values. This has the same effect as omitting the registration options property. If other options are specified at the same time, None is ignored.

VariableUserId

(MQPSC_VARIABLE_USER_ID)

When specified the identity of the subscriber (queue, queue manager and correlid) is not restricted to a single userid. This differs from the existing behavior of the queue manager that associates the userid of the original registration message with the subscriber's identity and from then on prevents any other user using that identity. If a new subscriber tries to use the same identity, the return code *MQRCCF_DUPLICATE_SUBSCRIPTION* is returned.

Any user can modify or deregister the subscription when they have suitable authority, avoiding the existing check that the userid must match that of the original subscriber.

To add this option to an existing subscription the command must come from the same userid as the original subscription itself.

If the subscription to be deregistered has VariableUserId set this must be set at deregister time to indicate which subscription is being deregistered. Otherwise, the userid of the

Deregister Subscriber command is used to identify the subscription. This is overridden, along with the other subscriber identifiers, if a subscription name is supplied.

The default, if this property is omitted, is that no registration options are set.

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is the queue manager name for the subscriber queue. It must match the QMgrName used in the original subscription.

If this property is omitted, the default is the ReplyToQMgr name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the name of the queue manager.

<QName> (MQPSC_Q_NAME)

The value is the name of the subscriber queue. It must match the QName used in the original subscription.

If this property is omitted, the default is the ReplyToQ name in the message descriptor (MQMD), which must not be blank.

<SubName> (MQPSC_SUBSCRIPTION_NAME)

If you specify SubName on a **Deregister Subscriber** command the SubName value takes precedence over all other identifier fields except the userid, unless VariableUserId is set on the subscription itself. If VariableUserId is not set, the **Deregister Subscriber** command succeeds only if the userid of the command message matches that of the subscription, if not the command fails with return code *MQRCCF_DUPLICATE_IDENTITY*.

If a subscription exists that matches the traditional identity of this command but has no SubName the **Deregister Subscriber** command fails with return code *MQRCCF_SUB_NAME_ERROR*. If an attempt is made to deregister a subscription that has a SubName using a command message that matches the traditional identity but with no SubName specified the command succeeds.

<SubUserData> (MQPSC_SUBSCRIPTION_USER_DATA)

This is a variable-length text string. The value is stored by the queue manager with the subscription but has no influence on the delivery of the publication to the subscriber. The value can be altered by re-registering to the same subscription with a new value. This attribute is for the use of the application.

SubUserData is returned in the Metatopic information (MQCACF_REG_SUB_USER_DATA) for a subscription, if SubUserData is present.

Example

Here is an example of NameValueData for a **Deregister Subscriber** command message. In this example, the sample application is deregistering its subscription to the topics which contain the latest score for all matches. The subscriber's identity, including the CorrelId, is taken from the defaults in the MQMD.

```
<psc>
  <Command>DeregSub</Command>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```

Publish message

The **Publish** command message is put to a queue, or from a queue manager to a subscriber, to publish information on a specified topic or topics.

Authority to put a message onto a queue and authority to publish information on a specified topic or topics is necessary.

If the user has authority to publish information on some, but not all, topics, only those topics are used to publish; a warning response indicates which topics are not used to publish.

If a subscriber has any matching subscriptions, the queue manager forwards the **Publish** message to the subscriber queues defined in the corresponding **Register Subscriber** command messages.

See [Queue Manager Response message](#) for details of the message descriptor (MQMD) parameters needed when sending a command message to the queue manager, and used when a queue manager forwards a publication to a subscriber.

The queue manager forwards the **Publish** message to other queue managers in the network that have matching subscriptions, unless it is a local publication.

Publication data, if any, is included in the body of the message. The data can be described in an <mcd> folder in the NameValueData field of the MQRFH2 header.

Properties

<Command> (MQPSC_COMMAND)

The value is `Publish` (MQPSC_PUBLISH).

This property must be specified.

<Topic> (MQPSC_TOPIC)

The value is a string that contains a topic that categorizes this publication. No wildcard characters are allowed.

You must add the topic to the namelist `SYSTEM.QPUBSUB.QUEUE.NAMELIST`, see [Adding a stream](#) for instructions on how to complete this task.

This property must be specified, and can optionally be repeated for as many topics as needed.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The subscription point on which the publication is published.

In WebSphere Event Broker V6, the value of the <SubPoint> property is the value of the Subscription Point attribute of the Publication node that is handling the publishing.

In IBM MQ V7.0.1, the value of the <SubPoint> property must match the name of a subscription point. See [Adding a subscription point](#).

<PubOpt> (MQPSC_PUBLICATION_OPTION)

The publication options property can take the following values:

RetainPub

(MQPSC_RETAIN_PUB)

The queue manager is to retain a copy of the publication. If this option is not set, the publication is deleted as soon as the queue manager has sent the publication to all its current subscribers.

IsRetainedPub

(MQPSC_IS_RETAINED_PUB)

(Can only be set by a queue manager.) This publication has been retained by the queue manager. The queue manager sets this option to notify a subscriber that this publication was published earlier and has been retained, provided that the subscription has been registered with the `InformIfRetained` option. It is set only in response to a `Register Subscriber` or `Request Update` command message. Retained publications that are sent directly to subscribers do not have this option set.

Local

(MQPSC_LOCAL)

This option tells the queue manager that this publication must not be sent to other queue managers. All subscribers that registered at this queue manager receive this publication if they have matching subscriptions.

OtherSubsOnly

(MQPSC_OTHER_SUBS_ONLY)

This option allows simpler processing of conference-type applications, where a publisher is also a subscriber to the same topic. It tells the queue manager not to send the publication to the publisher's subscriber queue even if it has a matching subscription. The publisher's subscriber queue consists of its QMgrName, QName, and optional CorrelId, as described in the following list.

CorrelAsId

(MQPSC_CORREL_ID_AS_IDENTITY)

The CorrelId in the MQMD (which must not be zero) is part of the publisher's subscriber queue, in applications where the publisher is also a subscriber.

None

(MQPSC_NONE)

All options take their default values. This has the same effect as omitting the publication options property. If other options are specified at the same time, None is ignored.

You can have more than one publication option by introducing additional <PubOpt> elements.

The default, if this property is omitted, is that no publication options are set.

<PubTime> (MQPSC_PUBLISH_TIMESTAMP)

The value is an optional publication timestamp set by the publisher. It is 16 characters long with format:

```
YYYYMMDDHHMMSSTH
```

using Universal Time. This information is not checked by the queue manager before being sent to the subscribers.

<SeqNum> (MQPSC_SEQUENCE_NUMBER)

The value is an optional sequence number set by the publisher.

It must be incremented by 1 with each publication. However, this is not checked by the queue manager, which merely transmits this information to subscribers.

If publications on the same topic are published to different interconnected queue managers, it is the responsibility of the publishers to ensure that sequence numbers, if used, are meaningful.

<QMGrName> (MQPSC_Q_MGR_NAME)

The value is a string containing the name of the queue manager for the publisher's subscriber queue, in applications where the publisher is also a subscriber (see OtherSubsOnly).

If this property is omitted, the default is the ReplyToQMGr name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the name of the queue manager.

<QName> (MQPSC_Q_NAME)

The value is a string containing the name of the publisher's subscriber queue, in applications where the publisher is also a subscriber (see OtherSubsOnly).

If this property is omitted, the default is the ReplyToQ name in the message descriptor (MQMD), which must not be blank if OtherSubsOnly is set.

Example

Here are some examples of *NameValueData* for a **Publish** command message.

The first example is for a publication sent by the match simulator in the sample application to indicate that a match has started.

```
<psc>
  <Command>Publish</Command>
  <Topic>Sport/Soccer/Event/MatchStarted</Topic>
</psc>
```

The second example is for a retained publication. The latest score in the match between Team1 and Team2 is published.

```
<psc>
  <Command>Publish</Command>
  <PubOpt>RetainPub</PubOpt>
  <Topic>Sport/Soccer/State/LatestScore/Team1 Team2</Topic>
</psc>
```

Register Subscriber message

The **Register Subscriber** command message is sent to a queue manager by a subscriber, or by another application on behalf of a subscriber, to indicate that it wants to subscribe to one or more topics at a subscription point. A message content filter can also be specified.

In publish/subscribe filter expressions, nesting parentheses causes performance to decrease exponentially. Avoid nesting parentheses to a depth greater than about 6.

The message is sent to SYSTEM.BROKER.CONTROL.QUEUE, which is the queue manager's control queue. Authority to put a message to this queue is required, in addition to access authority (set by the queue manager's system administrator) for the topic, or topics, in the subscription.

If the user has authority on some, but not all, topics, only those with authority are registered; a warning response indicates those that are not registered.

See “MQMD settings in command messages to the queue manager” on page 861 for details of the message descriptor (MQMD) parameters that are needed when sending a command message to the queue manager.

If the reply to queue is a temporary dynamic queue, the subscription is deregistered automatically by the queue manager when the queue is closed.

Properties

<Command> (MQPSC_COMMAND)

The value is RegSub (MQPSC_REGISTER_SUBSCRIBER). This property must be specified.

<Topic> (MQPSC_TOPIC)

The topic for which the subscriber wants to receive publications. Wildcard characters can be specified as part of the topic.

If you use the MQSC command **display sub** to examine the subscription created in this way, the value of the <Topic> tag is shown as the TOPICSTR property of the subscription.

This property is required, and can optionally be repeated for as many topics as needed.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is the subscription point to which the subscription is attached.

If this property is omitted, the default subscription point is used.

In WebSphere Event Broker V6, the value of the <SubPoint> property must match the value of the Subscription Point attribute of the Publication nodes that are subscribed to.

In IBM MQ V7.0.1, the value of the <SubPoint> property must match the name of a subscription point. See [Adding a subscription point](#).

<Filter> (MQPSC_FILTER)

The value is an SQL expression that is used as a filter on the contents of publication messages.

If a publication on the specified topic matches the filter, it is sent to the subscriber. This property corresponds to the Selection String that is used in MQSUB and MQOPEN calls. For more information, see [Selecting on the content of a message](#)

If this property is omitted, no content filtering takes place.

<RegOpt> (MQPSC_REGISTRATION_OPTION)

This Registration Options property can take the following values:

AddName

(MQPSC_ADD_NAME)

When specified for an existing subscription that matches the traditional identity of this Register Subscription command, but with no current SubName value, the SubName specified in this command is added to the subscription.

If AddName is specified the SubName field is mandatory, otherwise MQRCCF_REG_OPTIONS_ERROR is returned.

CorrelAsId

(MQPSC_CORREL_ID_AS_IDENTITY)

The CorrelId in the message descriptor (MQMD) is used when sending matching publications to the subscriber queue. The CorrelId must not be zero,

FullResp

(MQPSC_FULL_RESPONSE)

When specified all attributes of the subscription are returned in the response message, if the command does not fail.

FullResp is valid only when the command message refers to a single subscription. Therefore, only one topic is permitted in the command; otherwise the command fails with return code MQRCCF_REG_OPTIONS_ERROR.

InformIfRet

(MQPSC_INFORM_IF_RETAINED)

The queue manager informs the subscriber if a publication is retained when it sends a Publish message in response to a **Register Subscriber** or **Request Update** command message. The queue manager does this by including the IsRetainedPub publication option in the message.

JoinExcl

(MQPSC_JOIN_EXCLUSIVE)

This option indicates that the specified SubIdentity should be added as the exclusive member of the identity set for the subscription, and that no other identities can be added to the set.

If the identity has already joined 'shared' and is the sole entry in the set, the set is changed to an exclusive lock held by this identity. Otherwise, if the subscription currently has other identities in the identity set (with shared access) the command fails with return code MQRCCF_SUBSCRIPTION_IN_USE.

JoinShared

(MQPSC_JOIN_SHARED)

This option indicates that the specified SubIdentity should be added to the identity set for the subscription.

If the subscription is currently locked exclusively (using the JoinExcl option), the command fails with return code MQRCCF_SUBSCRIPTION_LOCKED, unless the identity that has the subscription locked is the same identity as that in this command message. In this case the lock is automatically modified to a shared lock.

Local

(MQPSC_LOCAL)

The subscription is local and is not distributed to other queue managers in the network. Publications made at other queue managers are not delivered to this subscriber, unless it also has a corresponding global subscription.

NewPubsOnly

(MQPSC_NEW_PUBS_ONLY)

Retained publications that exist at the time the subscription is registered are not sent to the subscriber; only new publications are sent.

If a subscriber re-registers and changes this option so that it is no longer set, a publication that has already been sent to it might be sent again.

NoAlter

(MQPSC_NO_ALTER)

The attributes of an existing matching subscription is not changed.

When a subscription is being created, this option is ignored. All other options specified apply to the new subscription.

If a SubIdentity also has one of the join options (JoinExcl or JoinShared) specified, the identity is added to the identity set regardless of whether NoAlter is specified.

None

(MQPSC_NONE)

All registration options take their default values.

If the subscriber is already registered, its options are reset to their default values (note that this does *not* have the same affect as omitting the registration options property), and the subscription expiry is updated from the MQMD of the **Register Subscriber** message.

If other registration options are specified at the same time, None is ignored.

NonPers

(MQPSC_NON_PERSISTENT)

Publications matching this subscription are delivered to the subscriber as non-persistent messages.

Pers

(MQPSC_PERSISTENT)

Publications matching this subscription are delivered to the subscriber as persistent messages.

PersAsPub

(MQPSC_PERSISTENT_AS_PUBLISH)

Publications matching this subscription are delivered to the subscriber with the persistence specified by the publisher. This is the default behavior.

PersAsQueue

(MQPSC_PERSISTENT_AS_Q)

Publications matching this subscription are delivered to the subscriber with the persistence specified on the subscriber queue.

PubOnReqOnly

(MQPSC_PUB_ON_REQUEST_ONLY)

The queue manager does not send publications to the subscriber, except in response to a **Request Update** command message.

VariableUserId

(MQPSC_VARIABLE_USER_ID)

When specified the identity of the subscriber (queue, queue manager and correlid) is not restricted to a single userid. This differs from the existing behavior of the queue manager that associates the userid of the original registration message with the subscriber's identity and from then on prevents any other user using that identity. If a new subscriber tries to use the same identity **MQRCCF_DUPLICATE_SUBSCRIPTION** is returned.

This allows any user to modify or deregister the subscription if the user has suitable authority. There is therefore no need to check that the userid matches that of the original subscriber.

To add this option to an existing subscription the command must come from the same userid as the original subscription itself.

If the subscription of the **Request Update** command has VariableUserId set, this must be set at request update time to indicate which subscription is referred to. Otherwise, the userid of

the **Request Update** command is used to identify the subscription. This is overridden, along with the other subscriber identifiers, if a subscription name is supplied.

If a **Register Subscriber** command message without this option set refers to an existing subscription which has this option set, the option is removed from this subscription and the userid of the subscription is now fixed. If there already exists a subscriber which has the same identity (queue, queue manager and correlation identifier) but with a different user ID associated to it, the command fails with return code *MQRCCF_DUPLICATE_IDENTITY* because there can only be one userid associated with a subscriber identity.

If the registration options property is omitted and the subscriber is already registered, its registration options are not changed and the subscription expiry is updated from the MQMD of the **Register Subscriber** message.

If the subscriber is not already registered, a new subscription is created with all registration options taking their default values.

The default values are PersAsPub and no other options set.

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is the name of the queue manager for the subscriber queue, to which matching publications are sent by the queue manager.

If this property is omitted, the default is the ReplyToQMgr name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the queue manager's QMgrName.

<QName> (MQPSC_Q_NAME)

The value is the name of the subscriber queue, to which matching publications are sent by the queue manager.

If this property is omitted, the default is the ReplyToQ name in the message descriptor (MQMD), which must not be blank in this case.

If the queue is a temporary dynamic queue, nonpersistent delivery of publications (NonPers) must be specified in the <RegOpt> property.

If the queue is a temporary dynamic queue, the subscription is deregistered automatically by the queue manager when the queue is closed.

<SubName> (MQPSC_SUBSCRIPTION_NAME)

This is a name given to a particular subscription. You can use it instead of the queue manager, queue and optional CorrelId to refer to a subscription.

If a subscription already exists with this **SubName** , any other attributes of the subscription (Topic, QMgrName, QName, CorrelId, UserId, RegOpts, UserSubData, and Expiry) are overridden with the attributes, if specified, that are passed in the new **Register Subscriber** command message. However, if **SubName** is used with no QName field specified, and a ReplyToQ is specified in the MQMD header, the subscriber queue is changed to be the ReplyToQ.

If a subscription that matches the traditional identity of this command already exists, but has no **SubName** , the Registration command fails with return code *MQRCCF_DUPLICATE_SUBSCRIPTION*, unless the **AddName** option is specified.

If you try to alter an existing named subscription by using another **Register Subscriber** command that specifies the same **SubName** , and the values of Topic, QMgrName, QName, and CorrelId in the new command match a different existing subscription, with or without a SubName defined, the command fails with return code *MQRCCF_DUPLICATE_SUBSCRIPTION*. This prevents two subscription names referring to the same subscription.

<SubIdentity> (MQPSC_SUBSCRIPTION_IDENTITY)

This string is used to represent an application with an interest in a subscription. It is a variable-length character string with a maximum length of 64 characters, and is optional. The queue manager maintains a set of subscriber identities for each subscription. Each subscription can allow its identity set to contain only one identity, or an unlimited number of identities (see the **JoinShared** and **JoinExcl** options).

A subscribe command that specifies the **JoinShared** or **JoinExcl** option adds the **SubIdentity** to the subscription's identity set, if it is not already there and if the existing set of identities allows such an action; that is, no other subscriber has joined exclusively or the identity set is empty.

Any alteration of the subscription's attributes as the result of a **Register Subscriber** command in which a **SubIdentity** is specified, only succeeds if it would be the only member of the set of identities for this subscription. Otherwise the command fails with return code *MQRCCF_SUBSCRIPTION_IN_USE*. This prevents a subscription's attributes from changing without other interested subscribers being aware.

If you specify a character string that is longer than 64 characters, the command fails with return code *MQRCCF_SUB_IDENTITY_ERROR*.

<SubUserData> (MQPSC_SUBSCRIPTION_USER_DATA)

This is a variable-length text string. The value is stored by the queue manager with the subscription, but has no influence on publication delivery to the subscriber. The value can be altered by re-registering to the same subscription with a new value. This attribute is there for the use of the application.

The **SubUserData** is returned in the Metatopic information (*MQCACF_REG_SUB_USER_DATA*) for a subscription if present.

If you specify more than one of the registration option values *NonPers*, *PersAsPub*, *PersAsQueue*, and *Pers*, then only the last one is used. You cannot combine these options in an individual subscription.

Example

Here is an example of *NameValueData* for a **Register Subscriber** command message. In the sample application, the results service uses this message to register a subscription to the topics containing the latest scores in all matches, with the 'Persistent as publish' option set. The subscriber's identity, including the *CorrelId*, is taken from the defaults in the MQMD.

```
<psc>
  <Command>RegSub</Command>
  <RegOpt>PersAsPub</RegOpt>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```

Request Update message

The **Request Update** command message is sent from a subscriber to a queue manager, to request the current retained publications for the specified topic and subscription point that match the given (optional) filter.

This message is sent to *SYSTEM.BROKER.CONTROL.QUEUE*, the queue manager's control queue. Authority to put a message to this queue is required, in addition to access authority for the topic in the request update; this is set by the queue manager's system administrator.

This command is normally used if the subscriber specified the option *PubOnReqOnly* when it registered. If the queue manager has any matching retained publications, they are sent to the subscriber. If the queue manager has no matching retained publications, the request fails with return code *MQRCCF_NO_RETAINED_MSG*. The requester must have previously registered a subscription with the same *Topic*, *SubPoint*, and *Filter* values.

Properties

<Command> (MQPSC_COMMAND)

The value is *ReqUpdate* (*MQPSC_REQUEST_UPDATE*). This property must be specified.

<Topic> (MQPSC_TOPIC)

The value is the topic that the subscriber is requesting; wildcard characters are allowed.

This property must be specified, but only one occurrence is allowed in this message.

<SubPoint> (MQPSC_SUBSCRIPTION_POINT)

The value is the subscription point to which the subscription is attached.

If this property is omitted, the default subscription point is used.

<Filter> (MQPSC_FILTER)

The value is an ESQL expression that is used as a filter on the contents of publication messages. If a publication on the specified topic matches the filter, it is sent to the subscriber.

The <Filter> property should have the same value as that specified on the original subscription for which you are now requesting an update.

If this property is omitted, no content filtering takes place.

<RegOpt> (MQPSC_REGISTRATION_OPTION)

The registration options property can take the following value:

CorrelAsId

(MQPSC_CORREL_ID_AS_IDENTITY)

The CorrelId in the message descriptor (MQMD), which must not be zero, is used when sending matching publications to the subscriber queue.

None

(MQPSC_NONE)

All options take their default values. This has the same effect as omitting the <RegOpt> property. If other options are specified at the same time, None is ignored.

VariableUserId

(MQPSC_VARIABLE_USER_ID)

When specified the identity of the subscriber (queue, queue manager, and correlid) is not restricted to a single userid. This differs from the existing behavior of the queue manager that associates the userid of the original registration message with the subscriber's identity and from then on prevents any other user using that identity. If a new subscriber tries to use the same identity, the command fails with return code *MQRCCF_DUPLICATE_SUBSCRIPTION*.

This allows any user to modify or deregister the subscription when they have suitable authority. Therefore, there is no need to check that the userid matches that of the original subscriber.

To add this option to an existing subscription, the command must come from the same userid as the original subscription.

If the subscription of the **Request Update** command has VariableUserId set, this must be set at request update time to indicate which subscription is referred to. Otherwise, the userid of the **Request Update** command is used to identify the subscription. This is overridden, along with the other subscriber identifiers, if a subscription name is supplied.

The default, if this property is omitted, is that no registration options are set.

<QMgrName> (MQPSC_Q_MGR_NAME)

The value is the name of the queue manager for the subscriber queue, to which the matching retained publication is sent by the queue manager.

If this property is omitted, the default is the ReplyToQMgr name in the message descriptor (MQMD). If the resulting name is blank, it defaults to the queue manager's QMgrName.

<QName> (MQPSC_Q_NAME)

The value is the name of the subscriber queue, to which the matching retained publication is sent by the queue manager.

If this property is omitted, the default is the ReplyToQ name in the message descriptor (MQMD), which must not be blank in this case.

<SubName> (MQPSC_SUBSCRIPTION_NAME)

This is a name given to a particular subscription. If specified on a **Request Update** command the SubName value takes precedence over all other identifier fields except the userid, unless

VariableUserId is set on the subscription itself. If VariableUserId is not set, the *Request Update* command succeeds only if the userid of the command message matches that of the subscription. If the userid of the command message does not match that of the subscription, the command fails with return code *MQRCCF_DUPLICATE_IDENTITY*.

If VariableUserId is set, and the userid differs from that of the subscription, the command succeeds if the userid of the new command message has authority to browse the stream queue and put to the subscriber queue of the subscription. Otherwise, the command fails with return code *MQRCCF_NOT_AUTHORIZED*.

If a subscription exists that matches the traditional identity of this command, but has no SubName, the **Request Update** command fails with return code *MQRCCF_SUB_NAME_ERROR*.

If an attempt is made to request an update for a subscription that has a SubName using a command message that matches the traditional identity, but with no SubName specified, the command succeeds.

Example

Here is an example of NameValueData for a **Request Update** command message. In the sample application, the results service uses this message to request retained publications containing the latest scores for all teams. The subscriber's identity, including the CorrelId, is taken from the defaults in the MQMD.

```
<psc>
  <Command>ReqUpdate</Command>
  <RegOpt>CorrelAsId</RegOpt>
  <Topic>Sport/Soccer/State/LatestScore/#</Topic>
</psc>
```

Queue Manager Response message

A **Queue Manager Response** message is sent from a queue manager to the ReplyToQ of a publisher or a subscriber, to indicate the success or failure of a command message received by the queue manager if the command message descriptor specified that a response is required.

The response message is contained within the NameValueData field of the MQRFH2 header, in a <pscr> folder.

In the case of a warning or error, the response message contains the <psc> folder from the command message as well as the <pscr> folder. The message data, if any, is not contained in the queue manager response message. In the case of an error, none of the message that caused an error has been processed; in the case of a warning, some of the message might have been processed successfully.

If there is a failure sending a response:

- For publication messages, the queue manager tries to send the response to the IBM MQ dead-letter queue if the MQPUT fails. This allows the publication to be sent to subscribers even if the response cannot be sent back to the publisher.
- For other messages, or if the publication response cannot be sent to the dead-letter queue, an error is logged and the command message is normally rolled back. Whether this happens depends on how the MQInput node has been configured.

Properties

<Completion> (MQPSCR_COMPLETION)

The completion code, which can take one of three values:

ok

Command completed successfully

warning

Command completed but with warning

error

Command failed

<Response> (MQPSCR_RESPONSE)

The response to a command message, if that command produced a completion code of warning or error. It contains a <Reason> property, and might contain other properties that indicate the cause of the warning or error.

In the case of one or more errors, there is only one response folder, indicating the cause of the first error only. In the case of one or more warnings, there is a response folder for each warning.

<Reason> (MQPSCR_REASON)

The reason code qualifying the completion code, if the completion code is a warning or error. It is set to one of the error codes listed in the following example. The <Reason> property is contained within a <Response> folder. The reason code can be followed by any valid property from the <psc> folder (for example, a topic name), indicating the cause of the error or warning. If you get a reason code of ????, check the data for correctness, for example, matching angled brackets (< >).

Examples

Here are some examples of NameValueData in a **Queue Manager Response** message. A successful response might be the following:

```
<pscr>
  <Completion>ok</Completion>
</pscr>
```

Here is an example of a failure response; the failure is a filter error. The first NameValueData string contains the response; the second contains the original command.

```
<pscr>
  <Completion>error</Completion>
  <Response>
    <Reason>3150</Reason>
  </Reponse>
</pscr>

<psc>
  ...
  command message (to which
  the queue manager is responding)
  ...
</psc>
```

Here is an example of a warning response (due to unauthorized topics). The first NameValueData string contains the response; the second NameValueData string contains the original command.

```
<pscr>
  <Completion>warning</Completion>
  <Response>
    <Reason>3081</Reason>
    <Topic>topic1</Topic>
  </Reponse>
  <Response>
    <Reason>3081</Reason>
    <Topic>topic2</Topic>
  </Reponse>
</pscr>

<psc>
  ...
  command message (to which
  the queue manager is responding)
  ...
</psc>
```

Publish/subscribe reason codes

These reason codes might be returned in the Reason field of a publish/subscribe response <psc> folder. Constants that can be used to represent these codes in the C or C++ programming languages are also listed.

The MQRC_ constants require the IBM MQ cmqc.h header file. The MQRCCF_ constants require the IBM MQ cmqcf.h header file (apart from *MQRCCF_FILTER_ERROR* and *MQRCCF_WRONG_USER*, which require the cmqpsc.h header file).

Reason code and text	Explanation	Issued by
2336 MQRC_RFH_COMMAND_ERROR	Valid values for the <Command> field of a <psc> folder are: RegSub, DeregSub, Publish, DeletePub, and ReqUpdate. Any other values result in this error code being issued.	Any command
2337 MQRC_RFH_PARM_ERROR	The <psc> and <mcd> folders both have a set of valid parameters that can be specified within them. Check the descriptions of these folders and ensure that you have not specified incorrect parameters.	Any command
2338 MQRC_RFH_DUPLICATE_PARM	Some parameters (for example, Topic) within a <psc> folder can be repeated, but others (for example, Command) cannot be repeated. Check that you have not duplicated a non-repeatable parameter.	Any command
2339 MQRC_RFH_PARM_MISSING	Some parameters within <psc> or <mcd> folders are optional and can be omitted; some are mandatory and must not be omitted. Check that you have included all mandatory parameters within your <psc> and <mcd> folders.	Any command
2551 MQRC_SELECTION_NOT_AVAILABLE	No extended message selection provider was available to determine which subscribers with a filter specified should receive the publication.	Publish, Register Subscriber, and Request Update
	No extended message selection provider was available to handle the filter of the specified subscriber.	Register Subscriber and Request Update
2554 MQRC_CONTENT_ERROR	An extended message selection provider found an error in the current or retained publication.	Publish and Request Update
3008 MQRCCF_COMMAND_FAILED	An internal error occurred which prevented the command from executing correctly. The error might occur if the command is reissued. The system event log for the queue manager contains information which should be used when reporting the problem to IBM.	Any command

Reason code and text	Explanation	Issued by
3072 MQRCCF_TOPIC_ERROR	One or more of the values you supplied for the Topic parameter are incorrect. Check that your values for Topic conform to the specified restrictions.	Any command
3073 MQRCCF_NOT_REGISTERED	The combination of SubPoint, Topic, and Filter that you specified on your DeregSub or ReqUpdate command was either not a combination with which you had previously registered or, for the DeregSub command if the DeregAll option was specified, one of the SubPoint, Topic, or Filter properties was not used to deregister any subscription.	Deregister Subscriber and Request Update commands
3074 MQRCCF_Q_MGR_NAME_ERROR	The specified queue manager was not valid, or the queue manager was not available or did not exist.	Deregister Subscriber, Publish, Register Subscriber, and Request Update commands
3076 MQRCCF_Q_NAME_ERROR	The specified queue name was not valid, or the queue did not exist on the specified queue manager.	Deregister Subscriber, Publish, Register Subscriber, and Request Update commands
3077 MQRCCF_NO_RETAINED_MSG	There were no retained messages for the topic you specified. This might or might not be an error, depending on the design of your application program.	Request Update command
3079 MQRCCF_INCORRECT_Q	RegSub, DeregSub, and ReqUpdate commands are always sent to the SYSTEM.BROKER.CONTROL.QUEUE queue of the queue manager for which they are intended. Publish and Delete Publication commands are sent to the input queue for the particular publish/subscribe message flow for which they are intended; this is determined when the message flow is designed. This error code is returned if a command is sent to the wrong queue.	Any command
3080 MQRCCF_CORREL_ID_ERROR	You have specified CorrelAsId as one of your RegOpt parameters. However, the CorrelId field of the MQMD does not contain a valid correlation identifier (that is, it is set to MQCI_NONE).	Deregister Subscriber and Register Subscriber commands

Reason code and text	Explanation	Issued by
3081 MQRCCF_NOT_AUTHORIZED	You are not authorized to perform the requested action. Authorization settings for the queue manager are handled by the system administrator using the Topics Hierarchy editor.	Publish and Register Subscriber commands
3083 MQRCCF_REG_OPTIONS_ERROR	You have specified an unrecognized RegOpt parameter in the <psc> folder that contains your RegSub or DeregSub command.	Deregister Subscriber and Register Subscriber commands
3084 MQRCCF_PUB_OPTIONS_ERROR	You have specified an unrecognized PubOpt parameter in the <psc> folder that contains your Publish command.	Publish command
3087 MQRCCF_DEL_OPTIONS_ERROR	You have specified an unrecognized DelOpt parameter in the <psc> folder that contains your DeletePub command.	Delete Publication command
3150 MQRCCF_FILTER_ERROR	The value specified for the Filter parameter is not valid. Check the section that describes the valid syntax for filter expressions and ensure that your expression conforms.	Deregister Subscriber, Register Subscriber, and Request Update commands
3151 MQRCCF_WRONG_USER	A subscription that matches the one specified already exists; however, it was registered by a different user. A subscription can only be changed or deregistered by the user who originally registered it.	Deregister Subscriber, Register Subscriber, and Request Update commands
3152 MQRCCF_DUPLICATE_SUBSCRIPTION	A matching subscription already exists with a different subscription name.	
3153 MQRCCF_SUB_NAME_ERROR	Either the format of the subscription name is not valid, or a matching subscription already exists with no subscription name.	
3154 MQRCCF_SUB_IDENTITY_ERROR	The subscription identity parameter is in error. Either the supplied value exceeds the maximum length allowed, or the subscription identity is not currently a member of the subscription's identity set and a Join registration option was not specified.	
3155 MQRCCF_SUBSCRIPTION_IN_USE	An attempt to modify or deregister a subscription was attempted by a member of the identity set when it was not the only member of this set.	
3156 MQRCCF_SUBSCRIPTION_LOCKED	The subscription is currently exclusively locked by another identity.	

Reason code and text	Explanation	Issued by
3157 MQRCCF_ALREADY_JOINED	A Join registration option was specified but the subscriber identity was already a member of the subscription's identity set.	

MQMD settings in command messages to the queue manager

Applications that send command messages to the queue manager use the following settings of fields in the message descriptor (MQMD). Fields that are left as the default value, or can be set to any valid value in the usual way, are not listed here.

Report

See `MsgType` and `CorrelId`.

MsgType

`MsgType` should be set to either `MQMT_REQUEST` or `MQMT_DATAGRAM`. `MQRC_MSG_TYPE_ERROR` will be returned if `MsgType` is not set to one of these values.

`MsgType` should be set to `MQMT_REQUEST` for a command message if a response is always required. The `MQRO_PAN` and `MQRO_NAN` flags in the `Report` field are not significant in this case.

If `MsgType` is set to `MQMT_DATAGRAM`, responses depend on the setting of the `MQRO_PAN` and `MQRO_NAN` flags in the `Report` field:

- `MQRO_PAN` alone means that the queue manager sends a response only if the command succeeds.
- `MQRO_NAN` alone means that the queue manager sends a response only if the command fails.
- If a command completes with a warning, a response is sent if either `MQRO_PAN` or `MQRO_NAN` is set.
- `MQRO_PAN` + `MQRO_NAN` means that the queue manager sends a response whether the command succeeds or fails. This has the same effect from the queue manager's perspective as setting `MsgType` to `MQMT_REQUEST`.
- If neither `MQRO_PAN` nor `MQRO_NAN` is set, no response is ever sent.

Format

Set to `MQFMT_RF_HEADER_2`

MsgId

This field is normally set to `MQMI_NONE`, so that the queue manager generates a unique value.

CorrelId

This field can be set to any value. If the sender's identity includes a `CorrelId`, specify this value, together with `MQRO_PASS_CORREL_ID` in the `Report` field, to ensure that it is set in all response messages sent by the queue manager to the sender.

ReplyToQ

This field defines the queue to which responses, if any, are to be sent. This might be the sender's queue; this has the advantage that the `QName` parameter can be omitted from the message. If, however, responses are to be sent to a different queue, the `QName` parameter is needed.

ReplyToQMgr

This field defines the queue manager for responses. If you leave this field blank (the default value), the local queue manager puts its own name in this field.

MQMD settings for publications forwarded by a queue manager

A queue manager uses these settings of fields in the message descriptor (MQMD) when it sends a publication to a subscriber. All other fields in the MQMD are set to their default values.

Report

`Report` is set to `MQRO_NONE`.

MsgType

MsgType is set to MQMT_DATAGRAM.

Expiry

Expiry is set to the value in the Publish message received from the publisher. In the case of a retained message, the time outstanding is reduced by the approximate time that the message has been at the queue manager.

Format

Format is set to MQFMT_RF_HEADER_2

MsgId

MsgId is set to a unique value.

CorrelId

If CorrelId is part of the subscriber's identity, this is the value specified by the subscriber when registering. Otherwise, it is a non-zero value chosen by the queue manager.

Priority

Priority takes the value set by the publisher, or as resolved if the publisher specified MQPRI_PRIORITY_AS_Q_DEF.

Persistence

Persistence takes the value set by the publisher, or as resolved if the publisher specified MQPER_PERSISTENCE_AS_Q_DEF, unless specified otherwise in the Register Subscriber message for the subscriber to which this publication is being sent.

ReplyToQ

ReplyToQ is set to blanks.

ReplyToQMgr

ReplyToQMgr is set to the name of the queue manager.

UserIdentifier

UserIdentifier is the subscriber's user identifier, as set when the subscriber registered.

AccountingToken

AccountingToken is the subscriber's accounting token, as set when the subscriber first registered.

ApplIdentityData

ApplIdentityData is the subscriber's application identity data, as set when the subscriber first registered.

PutApplType

PutApplType is set to MQAT_BROKER.

PutApplName

PutApplName is set to the first 28 characters of the name of the queue manager.

PutDate

PutDate is the date when the message was put to the subscriber's queue.

PutTime

PutTime is the time when the message was put to the subscriber's queue.

ApplOriginData

ApplOriginData is set to blanks.

MQMD settings in queue manager response messages

A queue manager uses these settings of fields in the message descriptor (MQMD) when sending a reply to a publication message. All other fields in the MQMD are set to their default values.

Report

Report is set to all zeros.

MsgType

MsgType is set to MQMT_REPLY.

Format

Format is set to MQFMT_RF_HEADER_2

MsgId

The setting of MsgId depends on the Report options in the original command message. By default, it is set to MQMI_NONE, so that the queue manager generates a unique value.

CorrelId

The setting of CorrelId depends on the Report options in the original command message. By default, this means that the CorrelId is set to the same value as the MsgId of the command message. This can be used to correlate commands with their responses.

Priority

Priority is set to the same value as in the original command message.

Persistence

Persistence is set to the value set in the original command message.

Expiry

Expiry is set to the same value as in the original command message received by the queue manager.

PutApplType

PutApplType is set to MQAT_BROKER.

PutApplName

PutApplName is set to the first 28 characters of name of the queue manager.

Other context fields are set as if generated with MQPMO_PASS_IDENTITY_CONTEXT.

Machine encodings

This section describes the structure of the *Encoding* field in the message descriptor.

See [“MQMD - Message descriptor” on page 396](#) for a summary of the fields in the structure.

The *Encoding* field is a 32-bit integer that is divided into four separate subfields; these subfields identify:

- The encoding used for binary integers
- The encoding used for packed-decimal integers
- The encoding used for floating-point numbers
- Reserved bits

Each subfield is identified by a bit mask that has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined:

MQENC_INTEGER_MASK

Mask for binary-integer encoding.

This subfield occupies bit positions 28 through 31 within the *Encoding* field.

MQENC_DECIMAL_MASK

Mask for packed-decimal-integer encoding.

This subfield occupies bit positions 24 through 27 within the *Encoding* field.

MQENC_FLOAT_MASK

Mask for floating-point encoding.

This subfield occupies bit positions 20 through 23 within the *Encoding* field.

MQENC_RESERVED_MASK

Mask for reserved bits.

This subfield occupies bit positions 0 through 19 within the *Encoding* field.

Binary-integer encoding

The following values are valid for the binary-integer encoding:

MQENC_INTEGER_UNDEFINED

Binary integers are represented using an encoding that is undefined.

MQENC_INTEGER_NORMAL

Binary integers are represented in the conventional way:

- The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address
- The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address

MQENC_INTEGER_REVERSED

Binary integers are represented in the same way as MQENC_INTEGER_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_INTEGER_NORMAL.

Packed-decimal-integer encoding

The following values are valid for the packed-decimal-integer encoding:

MQENC_DECIMAL_UNDEFINED

Packed-decimal integers are represented using an encoding that is undefined.

MQENC_DECIMAL_NORMAL

Packed-decimal integers are represented in the conventional way:

- Each decimal digit in the printable form of the number is represented in packed decimal by a single hexadecimal digit in the range X'0' through X'9'. Each hexadecimal digit occupies four bits, and so each byte in the packed decimal number represents two decimal digits in the printable form of the number.
- The least significant byte in the packed-decimal number is the byte that contains the least significant decimal digit. Within that byte, the most significant four bits contain the least significant decimal digit, and the least significant four bits contain the sign. The sign is either X'C' (positive), X'D' (negative), or X'F' (unsigned).
- The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address.
- The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address.

MQENC_DECIMAL_REVERSED

Packed-decimal integers are represented in the same way as MQENC_DECIMAL_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_DECIMAL_NORMAL.

Floating-point encoding

The following values are valid for the floating-point encoding:

MQENC_FLOAT_UNDEFINED

Floating-point numbers are represented using an encoding that is undefined.

MQENC_FLOAT_IEEE_NORMAL

Floating-point numbers are represented using the standard IEEE ³ floating-point format, with the bytes arranged as follows:

³ The Institute of Electrical and Electronics Engineers

- The least significant byte in the mantissa has the highest address of any of the bytes in the number; the byte containing the exponent has the lowest address
- The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address

Details of the IEEE float encoding can be found in IEEE Standard 754.

MQENC_FLOAT_IEEE_REVERSED

Floating-point numbers are represented in the same way as MQENC_FLOAT_IEEE_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_FLOAT_IEEE_NORMAL.

MQENC_FLOAT_S390

Floating-point numbers are represented using the standard System/390 floating-point format; this is also used by System/370.

Constructing encodings

To construct a value for the *Encoding* field in MQMD, the relevant constants that describe the required encodings can be added together (do not add the same constant more than once), or combined using the bitwise OR operation (if the programming language supports bit operations).

Whichever method is used, combine only one of the MQENC_INTEGER_* encodings with one of the MQENC_DECIMAL_* encodings and one of the MQENC_FLOAT_* encodings.

Analyzing encodings

The *Encoding* field contains subfields; because of this, applications that need to examine the integer, packed decimal, or float encoding must use one of the techniques described.

Using bit operations

If the programming language supports bit operations, perform the following steps:

1. Select one of the following values, according to the type of encoding required:
 - MQENC_INTEGER_MASK for the binary integer encoding
 - MQENC_DECIMAL_MASK for the packed decimal integer encoding
 - MQENC_FLOAT_MASK for the floating point encoding
 Call the value A.
2. Combine the *Encoding* field with A using the bitwise AND operation; call the result B.
3. B is the encoding required, and can be tested for equality with each of the values that is valid for that type of encoding.

Using arithmetic

If the programming language *does not* support bit operations, perform the following steps using integer arithmetic:

1. Select one of the following values, according to the type of encoding required:
 - 1 for the binary integer encoding
 - 16 for the packed decimal integer encoding
 - 256 for the floating point encoding
 Call the value A.
2. Divide the value of the *Encoding* field by A ; call the result B.
3. Divide B by 16; call the result C.
4. Multiply C by 16 and subtract from B ; call the result D.

5. Multiply D by A ; call the result E.
6. E is the encoding required, and can be tested for equality with each of the values that is valid for that type of encoding.

Summary of machine architecture encodings

Encodings for machine architectures are shown in [Table 566 on page 866](#).

<i>Table 566. Summary of encodings for machine architectures</i>			
Machine architecture	Binary integer encoding	Packed-decimal integer encoding	Floating-point encoding
IBM i	normal	normal	IEEE normal
Intel x86	reversed	reversed	IEEE reversed
PowerPC®	normal	normal	IEEE normal
System/390	normal	normal	System/390

Report options and message flags

This section describes the *Report* and *MsgFlags* fields that are part of the message descriptor MQMD specified on the MQGET, MQPUT, and MQPUT1 calls.

The topics in this section describe:

- The structure of the report field and how the queue manager processes it
- How an application analyzes the report field
- The structure of the message-flags field

For more information about the MQMD message descriptor, see [“MQMD - Message descriptor” on page 396](#).

Structure of the report field

This information describes the structure of the report field.

The *Report* field is a 32-bit integer that is divided into three separate subfields. These subfields identify:

- Report options that are rejected if the local queue manager does not recognize them
- Report options that are always accepted, even if the local queue manager does not recognize them
- Report options that are accepted only if certain other conditions are satisfied

Each subfield is identified by a bit mask that has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits in a subfield are not necessarily adjacent. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MQRO_REJECT_UNSUP_MASK

This mask identifies the bit positions within the *Report* field where report options that are not supported by the local queue manager cause the MQPUT or MQPUT1 call to fail with completion code MQCC_FAILED and reason code MQRC_REPORT_OPTIONS_ERROR.

This subfield occupies bit positions 3, and 11 through 13.

MQRO_ACCEPT_UNSUP_MASK

This mask identifies the bit positions within the *Report* field where report options that are not supported by the local queue manager are nevertheless accepted on the MQPUT or MQPUT1 calls. Completion code MQCC_WARNING with reason code MQRC_UNKNOWN_REPORT_OPTION are returned in this case.

This subfield occupies bit positions 0 through 2, 4 through 10, and 24 through 31.

The following report options are included in this subfield:

- MQRO_ACTIVITY
- MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQRO_DEAD_LETTER_Q
- MQRO_DISCARD_MSG
- MQRO_EXCEPTION
- MQRO_EXCEPTION_WITH_DATA
- MQRO_EXCEPTION_WITH_FULL_DATA
- MQRO_EXPIRATION
- MQRO_EXPIRATION_WITH_DATA
- MQRO_EXPIRATION_WITH_FULL_DATA
- MQRO_NAN
- MQRO_NEW_MSG_ID
- MQRO_NONE
- MQRO_PAN
- MQRO_PASS_CORREL_ID
- MQRO_PASS_MSG_ID

MQRO_ACCEPT_UNSUP_IF_XMIT_MASK

This mask identifies the bit positions within the *Report* field where report options that are not supported by the local queue manager are nevertheless accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ObjectQMgrName* and *ObjectName* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code MQCC_WARNING with reason code MQRC_UNKNOWN_REPORT_OPTION are returned if these conditions are satisfied, and MQCC_FAILED with reason code MQRC_REPORT_OPTIONS_ERROR if not.

This subfield occupies bit positions 14 through 23.

The following report options are included in this subfield:

- MQRO_COA
- MQRO_COA_WITH_DATA
- MQRO_COA_WITH_FULL_DATA
- MQRO_COD
- MQRO_COD_WITH_DATA
- MQRO_COD_WITH_FULL_DATA

If any options are specified in the *Report* field that the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *Report* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described previously are returned.

If MQCC_WARNING is returned, it is not defined which reason code is returned if other warning conditions exist.

The ability to specify and have accepted report options that are not recognized by the local queue manager is useful when sending a message with a report option that is recognized and processed by a *remote* queue manager.

Analyzing the report field

The *Report* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report must use one of the techniques described.

Using bit operations

If the programming language supports bit operations, perform the following steps:

1. Select one of the following values, according to the type of report to be checked:

- MQRO_COA_WITH_FULL_DATA for COA report
- MQRO_COD_WITH_FULL_DATA for COD report
- MQRO_EXCEPTION_WITH_FULL_DATA for exception report
- MQRO_EXPIRATION_WITH_FULL_DATA for expiration report

Call the value A.

On z/OS, use the MQRO_*_WITH_DATA values instead of the MQRO_*_WITH_FULL_DATA values.

2. Combine the *Report* field with A using the bitwise AND operation; call the result B.
3. Test B for equality with each value that is possible for that type of report.

For example, if A is MQRO_EXCEPTION_WITH_FULL_DATA, test B for equality with each of the following to determine what was specified by the sender of the message:

- MQRO_NONE
- MQRO_EXCEPTION
- MQRO_EXCEPTION_WITH_DATA
- MQRO_EXCEPTION_WITH_FULL_DATA

The tests can be performed in whatever order is most convenient for the application logic.

Use a similar method to test for the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options; select as the value A whichever of these two constants is appropriate, and then proceed as described previously.

Using arithmetic

If the programming language *does not* support bit operations, perform the following steps using integer arithmetic:

1. Select one of the following values, according to the type of report to be checked:

- MQRO_COA for COA report
- MQRO_COD for COD report
- MQRO_EXCEPTION for exception report
- MQRO_EXPIRATION for expiration report

Call the value A.

2. Divide the *Report* field by A ; call the result B.
3. Divide B by 8 ; call the result C.
4. Multiply C by 8 and subtract from B ; call the result D.
5. Multiply D by A ; call the result E.
6. Test E for equality with each value that is possible for that type of report.

For example, if A is MQRO_EXCEPTION, test E for equality with each of the following to determine what was specified by the sender of the message:

- MQRO_NONE
- MQRO_EXCEPTION
- MQRO_EXCEPTION_WITH_DATA
- MQRO_EXCEPTION_WITH_FULL_DATA

The tests can be performed in whatever order is most convenient for the application logic.

The following pseudocode illustrates this technique for exception report messages:

```
A = MQRO_EXCEPTION
B = Report/A
C = B/8
D = B - C*8
E = D*A
```

Use a similar method to test for the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options; select as the value A whichever of these two constants is appropriate, and then proceed as described previously, but replacing the value 8 in the previous steps by the value 2.

Structure of the message-flags field

This information describes the structure of the message-flags field.

The *MsgFlags* field is a 32-bit integer that is divided into three separate subfields. These subfields identify:

- Message flags that are rejected if the local queue manager does not recognize them
- Message flags that are always accepted, even if the local queue manager does not recognize them
- Message flags that are accepted only if certain other conditions are satisfied

Note: All subfields in *MsgFlags* are reserved for use by the queue manager.

Each subfield is identified by a bit mask that has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MQMF_REJECT_UNSUP_MASK

This mask identifies the bit positions within the *MsgFlags* field where message flags that are not supported by the local queue manager cause the MQPUT or MQPUT1 call to fail with completion code MQCC_FAILED and reason code MQRC_MSG_FLAGS_ERROR.

This subfield occupies bit positions 20 through 31.

The following message flags are included in this subfield:

- MQMF_LAST_MSG_IN_GROUP
- MQMF_LAST_SEGMENT
- MQMF_MSG_IN_GROUP
- MQMF_SEGMENT
- MQMF_SEGMENTATION_ALLOWED
- MQMF_SEGMENTATION_INHIBITED

MQMF_ACCEPT_UNSUP_MASK

This mask identifies the bit positions within the *MsgFlags* field where message flags that are not supported by the local queue manager are nevertheless accepted on the MQPUT or MQPUT1 calls. The completion code is MQCC_OK.

This subfield occupies bit positions 0 through 11.

MQMF_ACCEPT_UNSUP_IF_XMIT_MASK

This mask identifies the bit positions within the *MsgFlags* field where message flags that are not supported by the local queue manager are nevertheless accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ObjectQMGrName* and *ObjectName* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code MQCC_OK is returned if these conditions are satisfied, and MQCC_FAILED with reason code MQRC_MSG_FLAGS_ERROR if not.

This subfield occupies bit positions 12 through 19.

If there are flags specified in the *MsgFlags* field that the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *MsgFlags* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described previously are returned.

Data-conversion exit

This collection of topics describes the interface to the data-conversion exit, and the processing performed by the queue manager when data conversion is required.

For more information about data conversion, see *Data Conversion under IBM MQ* at <https://www.ibm.com/support/docview.wss?uid=swg27005729>.

The data-conversion exit is invoked as part of the processing of the MQGET call in order to convert the application message data to the representation required by the receiving application. Conversion of the application message data is optional; it requires the MQGMO_CONVERT option to be specified on the MQGET call.

The following subjects are described:

- The processing performed by the queue manager in response to the MQGMO_CONVERT option; see [“Conversion processing” on page 870](#).
- Processing conventions used by the queue manager when processing a built-in format; these conventions are recommended for user-written exits too. See [“Processing conventions” on page 872](#).
- Special considerations for converting report messages; see [“Conversion of report messages” on page 876](#).
- The parameters passed to the data-conversion exit; see [“MQ_DATA_CONV_EXIT - Data conversion exit” on page 888](#).
- A call that can be used from the exit to convert character data between different representations; see [“MQXCNV - Convert characters” on page 882](#).
- The data-structure parameter that is specific to the exit; see [“MQDXP - Data-conversion exit parameter” on page 876](#).

Conversion processing

This information describes the processing performed by the queue manager in response to the MQGMO_CONVERT option.

The queue manager performs the following actions if the MQGMO_CONVERT option is specified on the MQGET call, and there is a message to be returned to the application:

1. If one or more of the following is true, no conversion is necessary:
 - The message data is already in the character set and encoding required by the application issuing the MQGET call. The application must set the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter of the MQGET call to the values required, before issuing the call.

- The length of the message data is zero.
- The length of the *Buffer* parameter of the MQGET call is zero.

In these cases the message is returned without conversion to the application issuing the MQGET call; the *CodedCharSetId* and *Encoding* values in the *MsgDesc* parameter are set to the values in the control information in the message, and the call completes with one of the following combinations of completion code and reason code:

Completion code	Reason code
MQCC_OK	MQRC_NONE
MQCC_WARNING	MQRC_TRUNCATED_MSG_ACCEPTED
MQCC_WARNING	MQRC_TRUNCATED_MSG_FAILED

The following steps are performed only if the character set or encoding of the message data differs from the corresponding value in the *MsgDesc* parameter, and there is data to be converted:

2. If the *Format* field in the control information in the message has the value MQFMT_NONE, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

In all other cases conversion processing continues.

3. The message is removed from the queue and placed in a temporary buffer that is the same size as the *Buffer* parameter. For browse operations, the message is copied into the temporary buffer, instead of being removed from the queue.
4. If the message has to be truncated to fit in the buffer, the following is done:
 - If the MQGMO_ACCEPT_TRUNCATED_MSG option was *not* specified, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_TRUNCATED_MSG_FAILED.
 - If the MQGMO_ACCEPT_TRUNCATED_MSG option was specified, the completion code is set to MQCC_WARNING, the reason code is set to MQRC_TRUNCATED_MSG_ACCEPTED, and conversion processing continues.
5. If the message can be accommodated in the buffer without truncation, or the MQGMO_ACCEPT_TRUNCATED_MSG option was specified, the following is done:
 - If the format is a built-in format, the buffer is passed to the queue-manager's data-conversion service.
 - If the format is not a built-in format, the buffer is passed to a user-written exit with the same name as the format. If the exit cannot be found, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

If no error occurs, the output from the data-conversion service or from the user-written exit is the converted message, plus the completion code and reason code to be returned to the application issuing the MQGET call.

6. If the conversion is successful, the queue manager returns the converted message to the application. In this case, the completion code and reason code returned by the MQGET call are one of the following combinations:

Completion code	Reason code
MQCC_OK	MQRC_NONE
MQCC_WARNING	MQRC_TRUNCATED_MSG_ACCEPTED

However, if the conversion is performed by a user-written exit, other reason codes can be returned, even when the conversion is successful.

If the conversion fails, the queue manager returns the unconverted message to the application, with the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter set to the values in the control information in the message, and with completion code MQCC_WARNING.

Processing conventions

When converting a built-in format, the queue manager follows the processing conventions described.

User-written exits should also follow these conventions, although this is not enforced by the queue manager. The built-in formats converted by the queue manager are:

- MQFMT_ADMIN
- MQFMT_CICS (z/OS only)
- MQFMT_COMMAND_1
- MQFMT_COMMAND_2
- MQFMT_DEAD_LETTER_HEADER
- MQFMT_DIST_HEADER
- MQFMT_EVENT version 1
- MQFMT_EVENT version 2
- MQFMT_IMS
- MQFMT_IMS_VAR_STRING
- MQFMT_MD_EXTENSION
- MQFMT_PCF
- MQFMT_REF_MSG_HEADER
- MQFMT_RF_HEADER
- MQFMT_RF_HEADER_2
- MQFMT_STRING
- MQFMT_TRIGGER
- MQFMT_WORK_INFO_HEADER (z/OS only)
- MQFMT_XMIT_Q_HEADER

1. If the message expands during conversion, and exceeds the size of the *Buffer* parameter, the following is done:

- If the MQGMO_ACCEPT_TRUNCATED_MSG option was *not* specified, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_CONVERTED_MSG_TOO_BIG.
- If the MQGMO_ACCEPT_TRUNCATED_MSG option *was* specified, the message is truncated, the completion code is set to MQCC_WARNING, the reason code is set to MQRC_TRUNCATED_MSG_ACCEPTED, and conversion processing continues.

2. If truncation occurs (either before or during conversion), the number of valid bytes returned in the *Buffer* parameter can be *less than* the length of the buffer.

This can occur, for example, if a 4-byte integer or a DBCS character straddles the end of the buffer. The incomplete element of information is not converted, and those bytes in the returned message do not contain valid information. This can also occur if a message that was truncated before conversion shrinks during conversion.

If the number of valid bytes returned is less than the length of the buffer, the unused bytes at the end of the buffer are set to nulls.

3. If an array or string straddles the end of the buffer, as much of the data as possible is converted; only the particular array element or DBCS character which is incomplete is not converted; preceding array elements or characters are converted.

4. If truncation occurs (either before or during conversion), the length returned for the *DataLength* parameter is the length of the *unconverted* message before truncation.
 5. When strings are converted between single-byte character sets (SBCS), double-byte character sets (DBCS), or multi-byte character sets (MBCS), the strings can expand or contract.
 - In the PCF formats MQFMT_ADMIN, MQFMT_EVENT, and MQFMT_PCF, the strings in the MQCFST and MQCFSL structures expand or contract as necessary to accommodate the string after conversion.

For the string-list structure MQCFSL, the strings in the list might expand or contract by different amounts. If this happens, the queue manager pads the shorter strings with blanks to make them the same length as the longest string after conversion.

 - In the format MQFMT_REF_MSG_HEADER, the strings addressed by the *SrcEnvOffset*, *SrcNameOffset*, *DestEnvOffset*, and *DestNameOffset* fields expand or contract as necessary to accommodate the strings after conversion.
 - In the format MQFMT_RF_HEADER, the *NameValueString* field expands or contracts as necessary to accommodate the name/value pairs after conversion.
 - In structures with fixed field sizes, the queue manager allows strings to expand or contract within their fixed fields, provided that no significant information is lost. In this regard, trailing blanks and characters following the first null character in the field are treated as insignificant.
 - If the string expands, but only insignificant characters need to be discarded to accommodate the converted string in the field, the conversion succeeds and the call completes with MQCC_OK and reason code MQRC_NONE (assuming no other errors).
 - If the string expands, but the converted string requires significant characters to be discarded in order to fit in the field, the message is returned unconverted and the call completes with MQCC_WARNING and reason code MQRC_CONVERTED_STRING_TOO_BIG.

Note: Reason code MQRC_CONVERTED_STRING_TOO_BIG results in this case whether or not the MQGMO_ACCEPT_TRUNCATED_MSG option was specified.

 - If the string contracts, the queue manager pads the string with blanks to the length of the field.
 6. For messages consisting of one or more MQ header structures followed by user data, one or more of the header structures might be converted, while the remainder of the message is not. However, (with two exceptions) the *CodedCharSetId* and *Encoding* fields in each header structure always correctly indicate the character set and encoding of the data that follows the header structure.
- The two exceptions are the MQCIH and MQIIH structures, where the values in the *CodedCharSetId* and *Encoding* fields in those structures are not significant. For those structures, the data following the structure is in the same character set and encoding as the MQCIH or MQIIH structure itself.
7. If the *CodedCharSetId* or *Encoding* fields in the control information of the message being retrieved, or in the *MsgDesc* parameter, specify values that are undefined or not supported, the queue manager might ignore the error if the undefined or unsupported value does not need to be used in converting the message.
- For example, if the *Encoding* field in the message specifies an unsupported float encoding, but the message contains only integer data, or contains floating-point data that does not require conversion (because the source and target float encodings are identical), the error might not be diagnosed.
- If the error is diagnosed, the message is returned unconverted, with completion code MQCC_WARNING and one of the MQRC_SOURCE_*_ERROR or MQRC_TARGET_*_ERROR reason codes (as appropriate); the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to the values in the control information in the message.
- If the error is not diagnosed and the conversion completes successfully, the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are those specified by the application issuing the MQGET call.
8. In all cases, if the message is returned to the application unconverted the completion code is set to MQCC_WARNING, and the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to the values appropriate to the unconverted data. This is done for MQFMT_NONE also.

The *Reason* parameter is set to a code that indicates why the conversion could not be carried out, unless the message also had to be truncated; reason codes related to truncation take precedence over reason codes related to conversion. (To determine if a truncated message was converted, check the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter.)

When an error is diagnosed, either a specific reason code is returned, or the general reason code MQRC_NOT_CONVERTED. The reason code returned depends on the diagnostic capabilities of the underlying data-conversion service.

9. If completion code MQCC_WARNING is returned, and more than one reason code is relevant, the order of precedence is as follows:
 - a. The following reasons take precedence over all others; only one of the reasons in this group can arise:
 - MQRC_SIGNAL_REQUEST_ACCEPTED
 - MQRC_TRUNCATED_MSG_ACCEPTED
 - b. The order of precedence within the remaining reason codes is not defined.

10. On completion of the MQGET call:

- The following reason code indicates that the message was converted successfully:
 - MQRC_NONE
- The following reason codes indicate that the message *might* have been converted successfully (check the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to find out):
 - MQRC_MSG_MARKED_BROWSE_CO_OP
 - MQRC_TRUNCATED_MSG_ACCEPTED
- All other reason codes indicate that the message was not converted.

The following processing is specific to the built-in formats; it does not apply to user-defined formats:

11. With the exception of the following formats:

- MQFMT_ADMIN
- MQFMT_COMMAND_1
- MQFMT_COMMAND_2
- MQFMT_EVENT
- MQFMT_IMS_VAR_STRING
- MQFMT_PCF
- MQFMT_STRING

none of the built-in formats can be converted from or to character sets that do not have SBCS characters for the characters that are valid in queue names. If an attempt is made to perform such a conversion, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_SOURCE_CCSID_ERROR or MQRC_TARGET_CCSID_ERROR, as appropriate.

The Unicode character set UCS-2 is an example of a character set that does not have SBCS characters for the characters that are valid in queue names.

12. If the message data for a built-in format is truncated, fields within the message that contain lengths of strings, or counts of elements or structures, are *not* adjusted to reflect the length of the data actually returned to the application; the values returned for such fields within the message data are the values applicable to the message *before truncation*.

When processing messages such as a truncated MQFMT_ADMIN message, ensure that the application does not attempt to access data beyond the end of the data returned.

13. If the format name is MQFMT_DEAD_LETTER_HEADER, the message data begins with an MQDLH structure, possibly followed by zero or more bytes of application message data. The format, character set, and encoding of the application message data are defined by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQDLH structure at the start of the message. Because the MQDLH

structure and application message data can have different character sets and encodings, one, other, or both of the MQDLH structure and application message data might require conversion.

The queue manager converts the MQDLH structure first, as necessary. If conversion is successful, or the MQDLH structure does not require conversion, the queue manager checks the *CodedCharSetId* and *Encoding* fields in the MQDLH structure to see if conversion of the application message data is required. If conversion is required, the queue manager invokes the user-written exit with the name given by the *Format* field in the MQDLH structure, or performs the conversion itself (if *Format* is the name of a built-in format).

If the MQGET call returns a completion code of MQCC_WARNING, and the reason code is one of those indicating that conversion was not successful, one of the following applies:

- The MQDLH structure could not be converted. In this case the application message data will not have been converted either.
- The MQDLH structure was converted, but the application message data was not.

The application can examine the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter, and those in the MQDLH structure, in order to determine which of the previously applies.

14. If the format name is MQFMT_XMIT_Q_HEADER, the message data begins with an MQXQH structure, possibly followed by zero or more bytes of additional data. This additional data is usually the application message data (which may be of zero length), but there can also be one or more further MQ header structures present, at the start of the additional data.

The MQXQH structure must be in the character set and encoding of the queue manager. The format, character set, and encoding of the data following the MQXQH structure are given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMD structure contained *within* the MQXQH. For each subsequent MQ header structure present, the *Format*, *CodedCharSetId*, and *Encoding* fields in the structure describe the data that follows that structure; that data is either another MQ header structure, or the application message data.

If the MQGMO_CONVERT option is specified for an MQFMT_XMIT_Q_HEADER message, the application message data and certain of the MQ header structures are converted, *but the data in the MQXQH structure is not*. On return from the MQGET call, therefore:

- The values of the *Format*, *CodedCharSetId*, and *Encoding* fields in the *MsgDesc* parameter describe the data in the MQXQH structure, and *not* the application message data; the values are therefore *not* the same as those specified by the application that issued the MQGET call.

The effect of this is that an application that repeatedly gets messages from a transmission queue with the MQGMO_CONVERT option specified must reset the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to the values required for the application message data, before each MQGET call.

- The values of the *Format*, *CodedCharSetId*, and *Encoding* fields in the last MQ header structure present describe the application message data. If there are no other MQ header structures present, the application message data is described by these fields in the MQMD structure within the MQXQH structure. If conversion is successful, the values will be the same as those specified in the *MsgDesc* parameter by the application that issued the MQGET call.

If the message is a distribution-list message, the MQXQH structure is followed by an MQDH structure (plus its arrays of MQOR and MQPMR records), which in turn might be followed by zero or more further MQ header structures and zero or more bytes of application message data. Like the MQXQH structure, the MQDH structure must be in the character set and encoding of the queue manager, and it is not converted on the MQGET call, even if the MQGMO_CONVERT option is specified.

The processing of the MQXQH and MQDH structures described previously is primarily intended for use by message channel agents when they get messages from transmission queues.

Conversion of report messages

In general a report message can contain varying amounts of application message data, according to the report options specified by the sender of the original message. However, an activity report can contain data but without the report option mentioning `*_WITH_DATA` in the constant.

In particular, a report message can contain either:

1. No application message data

2. Some of the application message data from the original message

This occurs when the sender of the original message specifies `MQRO_*_WITH_DATA` and the message is longer than 100 bytes.

3. All the application message data from the original message

This occurs when the sender of the original message specifies `MQRO_*_WITH_FULL_DATA`, or specifies `MQRO_*_WITH_DATA` and the message is 100 bytes or shorter.

When the queue manager or message channel agent generates a report message, it copies the format name from the original message into the *Format* field in the control information in the report message. The format name in the report message might therefore imply a length of data that is different from the length actually present in the report message (cases 1 and 2 previously).

If the `MQGMO_CONVERT` option is specified when the report message is retrieved:

- For case 1 previously, the data-conversion exit is not invoked (because the report message has no data).
- For case 3 previously, the format name correctly implies the length of the message data.
- But for case 2 previously, the data-conversion exit is invoked to convert a message that is *shorter* than the length implied by the format name.

In addition, the reason code passed to the exit is usually `MQRC_NONE` (that is, the reason code does not indicate that the message has been truncated). This happens because the message data was truncated by the *sender* of the report message, and not by the receiver's queue manager in response to the `MQGET` call.

Because of these possibilities, the data-conversion exit must *not* use the format name to deduce the length of data passed to it; instead the exit must check the length of data provided, and be prepared to convert *less* data than the length implied by the format name. If the data can be converted successfully, completion code `MQCC_OK` and reason code `MQRC_NONE` must be returned by the exit. The length of the message data to be converted is passed to the exit as the *InBufferLength* parameter.

Product-sensitive programming interface

MQDXP - Data-conversion exit parameter

The `MQDXP` structure is a parameter that the queue manager passes to the data-conversion exit when the exit is invoked to convert the message data as part of the processing of the `MQGET` call. See the description of the `MQ_DATA_CONV_EXIT` call for details of the data conversion exit.

Character data in `MQDXP` is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in `MQDXP` is in the native machine encoding; this is given by `MQENC_NATIVE`.

Only the *DataLength*, *CompCode*, *Reason*, and *ExitResponse* fields in `MQDXP` can be changed by the exit; changes to other fields are ignored. However, the *DataLength* field *cannot* be changed if the message being converted is a segment that contains only part of a logical message.

When control returns to the queue manager from the exit, the queue manager checks the values returned in `MQDXP`. If the values returned are not valid, the queue manager continues processing as though the exit had returned `MQXDR_CONVERSION_FAILED` in *ExitResponse*; however, the queue manager ignores the values of the *CompCode* and *Reason* fields returned by the exit in this case, and uses instead

the values those fields had on *input* to the exit. The following values in MQDXP cause this processing to occur:

- *ExitResponse* field not MQXDR_OK and not MQXDR_CONVERSION_FAILED
- *CompCode* field not MQCC_OK and not MQCC_WARNING
- *DataLength* field less than zero, or *DataLength* field changed when the message being converted is a segment that contains only part of a logical message.

The following table summarizes the fields in the structure.

Table 567. Fields in MQDXP		
Field	Description	Topic
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>AppOptions</i>	Application options	AppOptions
<i>Encoding</i>	Numeric encoding required by application	Encoding
<i>CodedCharSetId</i>	Character set required by application	CodedCharSetId
<i>DataLength</i>	Length in bytes of message data	DataLength
<i>CompCode</i>	Completion code	CompCode
<i>Reason</i>	Reason code qualifying <i>CompCode</i>	Reason
<i>ExitResponse</i>	Response from exit	ExitResponse
<i>Hconn</i>	Connection handle	Hconn
<i>pEntryPoints</i>	Address of the MQIEP structure	pEntryPoints

Fields

The MQDXP structure contains the following fields; the fields are described in alphabetical order.

AppOptions

Type: MQLONG

This is a copy of the *Options* field of the MQGMO structure specified by the application issuing the MQGET call. The exit might need to examine these to ascertain whether the MQGMO_ACCEPT_TRUNCATED_MSG option was specified.

This is an input field to the exit.

CodedCharSetId

Type: MQLONG

This is the coded character-set identifier of the character set required by the application issuing the MQGET call; see the *CodedCharSetId* field in the MQMD structure for more details. If the application specifies the special value MQCCSI_Q_MGR on the MQGET call, the queue manager changes this to the actual character-set identifier of the character set used by the queue manager, before invoking the exit.

If the conversion is successful, the exit must copy this to the *CodedCharSetId* field in the message descriptor.

This is an input field to the exit.

CompCode

Type: MQLONG

When the exit is invoked, this contains the completion code that is returned to the application that issued the MQGET call, if the exit does nothing. It is always MQCC_WARNING, because either the message was truncated, or the message requires conversion and this has not yet been done.

On output from the exit, this field contains the completion code to be returned to the application in the *CompCode* parameter of the MQGET call; only MQCC_OK and MQCC_WARNING are valid. See the description of the *Reason* field for suggestions on how the exit can set this field on output.

This is an input/output field to the exit.

DataLength

Type: MQLONG

When the exit is invoked, this field contains the original length of the application message data. If the message was truncated to fit into the buffer provided by the application, the size of the message provided to the exit is *smaller* than the value of *DataLength*. The size of the message provided to the exit is always given by the *InBufferLength* parameter of the exit, irrespective of any truncation that has occurred.

Truncation is indicated by the *Reason* field having the value MQRC_TRUNCATED_MSG_ACCEPTED on input to the exit.

Most conversions do not need to change this length, but an exit can do so if necessary; the value set by the exit is returned to the application in the *DataLength* parameter of the MQGET call. However, this length *cannot* be changed if the message being converted is a segment that contains only part of a logical message. This is because changing the length would cause the offsets of later segments in the logical message to be incorrect.

Note that, if the exit wants to change the length of the data, be aware that the queue manager has already decided whether the message data fits into the application's buffer, based on the length of the *unconverted* data. This decision determines whether the message is removed from the queue (or the browse cursor moved, for a browse request), and is not affected by any change to the data length caused by the conversion. For this reason it is recommended that conversion exits do not cause a change in the length of the application message data.

If character conversion does imply a change of length, a string can be converted into another string with the same length in bytes, truncating trailing blanks, or padding with blanks as necessary.

The exit is not invoked if the message contains no application message data; hence *DataLength* is always greater than zero.

This is an input/output field to the exit.

Encoding

Type: MQLONG

Numeric encoding required by application.

This is the numeric encoding required by the application issuing the MQGET call; see the *Encoding* field in the MQMD structure for more details.

If the conversion is successful, the exit copies this to the *Encoding* field in the message descriptor.

This is an input field to the exit.

ExitOptions

Type: MQLONG

This is a reserved field; its value is 0.

ExitResponse

Type: MQLONG

Response from exit. This is set by the exit to indicate the success or otherwise of the conversion. It must be one of the following:

MQXDR_OK

Conversion was successful.

If the exit specifies this value, the queue manager returns the following to the application that issued the MQGET call:

- The value of the *CompCode* field on output from the exit
- The value of the *Reason* field on output from the exit
- The value of the *DataLength* field on output from the exit
- The contents of the exit's output buffer *OutBuffer*. The number of bytes returned is the lesser of the exit's *OutBufferLength* parameter, and the value of the *DataLength* field on output from the exit.

If the *Encoding* and *CodedCharSetId* fields in the exit's message descriptor parameter are *both* unchanged, the queue manager returns:

- The value of the *Encoding* and *CodedCharSetId* fields in the MQDXP structure on *input* to the exit.

If one or both of the *Encoding* and *CodedCharSetId* fields in the exit's message descriptor parameter has been changed, the queue manager returns:

- The value of the *Encoding* and *CodedCharSetId* fields in the exit's message descriptor parameter on output from the exit

MQXDR_CONVERSION_FAILED

Conversion was unsuccessful.

If the exit specifies this value, the queue manager returns the following to the application that issued the MQGET call:

- The value of the *CompCode* field on output from the exit
- The value of the *Reason* field on output from the exit
- The value of the *DataLength* field on *input* to the exit
- The contents of the exit's input buffer *InBuffer*. The number of bytes returned is given by the *InBufferLength* parameter

If the exit has altered *InBuffer*, the results are undefined.

ExitResponse is an output field from the exit.

Hconn

Type: MQHCONN

This is a connection handle which can be used on the MQXCNVC call. This handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

pEntryPoints

Type: PMQIEP

The address of an MQIEP structure through which MQI and DCI calls can be made.

Reason

Type: MQLONG

Reason code qualifying *CompCode*.

When the exit is invoked, this contains the reason code that is returned to the application that issued the MQGET call, if the exit chooses to do nothing. Among possible values are MQRC_TRUNCATED_MSG_ACCEPTED, indicating that the message was truncated in order fit into the buffer provided by the application, and MQRC_NOT_CONVERTED, indicating that the message requires conversion but that this has not yet been done.

On output from the exit, this field contains the reason to be returned to the application in the *Reason* parameter of the MQGET call; the following is recommended:

- If *Reason* had the value MQRC_TRUNCATED_MSG_ACCEPTED on input to the exit, the *Reason* and *CompCode* fields must not be altered, irrespective of whether the conversion succeeds or fails.

(If the *CompCode* field is not MQCC_OK, the application which retrieves the message can identify a conversion failure by comparing the returned *Encoding* and *CodedCharSetId* values in the message descriptor with the values requested; in contrast, the application cannot distinguish a truncated message from a message that fitted the buffer. For this reason, MQRC_TRUNCATED_MSG_ACCEPTED must be returned in preference to any of the reasons that indicate conversion failure.)

- If *Reason* had any other value on input to the exit:
 - If the conversion succeeds, *CompCode* must be set to MQCC_OK and *Reason* set to MQRC_NONE.
 - If the conversion fails, or the message expands and has to be truncated to fit in the buffer, *CompCode* must be set to MQCC_WARNING (or left unchanged), and *Reason* set to one of the values listed, to indicate the nature of the failure.

Note if the message after conversion is too large for the buffer, it must be truncated only if the application that issued the MQGET call specified the MQGMO_ACCEPT_TRUNCATED_MSG option:

- If it did specify that option, reason MQRC_TRUNCATED_MSG_ACCEPTED is returned.
- If it did not specify that option, the message is returned unconverted, with reason code MQRC_CONVERTED_MSG_TOO_BIG.

The reason codes listed are recommended for use by the exit to indicate the reason that conversion failed, but the exit can return other values from the set of MQRC_* codes if deemed appropriate. In addition, the range of values MQRC_APPL_FIRST through MQRC_APPL_LAST are allocated for use by the exit to indicate conditions that the exit wants to communicate to the application issuing the MQGET call.

Note: If the message cannot be converted successfully, the exit *must* return MQXDR_CONVERSION_FAILED in the *ExitResponse* field, in order to cause the queue manager to return the unconverted message. This is true regardless of the reason code returned in the *Reason* field.

MQRC_APPL_FIRST

(900, X'384') Lowest value for application-defined reason code.

MQRC_APPL_LAST

(999, X'3E7') Highest value for application-defined reason code.

MQRC_CONVERTED_MSG_TOO_BIG

(2120, X'848') Converted data too large for buffer.

MQRC_NOT_CONVERTED

(2119, X'847') Message data not converted.

MQRC_SOURCE_CCSID_ERROR

(2111, X'83F') Source coded character set identifier not valid.

MQRC_SOURCE_DECIMAL_ENC_ERROR

(2113, X'841') Packed-decimal encoding in message not recognized.

MQRC_SOURCE_FLOAT_ENC_ERROR

(2114, X'842') Floating-point encoding in message not recognized.

MQRC_SOURCE_INTEGER_ENC_ERROR

(2112, X'840') Source integer encoding not recognized.

MQRC_TARGET_CCSID_ERROR

(2115, X'843') Target coded character set identifier not valid.

MQRC_TARGET_DECIMAL_ENC_ERROR

(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

MQRC_TARGET_FLOAT_ENC_ERROR

(2118, X'846') Floating-point encoding specified by receiver not recognized.

MQRC_TARGET_INTEGER_ENC_ERROR

(2116, X'844') Target integer encoding not recognized.

MQRC_TRUNCATED_MSG_ACCEPTED

(2079, X'81F') Truncated message returned (processing completed).

This is an input/output field to the exit.

StrucId

Type: MQCHAR4

Structure identifier. The value must be:

MQDXP_STRUC_ID

Identifier for data conversion exit parameter structure.

For the C programming language, the constant MQDXP_STRUC_ID_ARRAY is also defined; this has the same value as MQDXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version

Type: MQLONG

Structure version number. The value must be:

MQDXP_VERSION_1

Version number for data-conversion exit parameter structure.

The following constant specifies the version number of the current version:

MQDXP_CURRENT_VERSION

Current version of data-conversion exit parameter structure.

Note: When a new version of this structure is introduced, the layout of the existing part is not changed. The exit must therefore check that the *Version* field is equal to or greater than the lowest version which contains the fields that the exit needs to use.

This is an input field to the exit.

C declaration

```
typedef struct tagMQDXP MQDXP;
struct tagMQDXP {
    MQCHAR4  StrucId;           /* Structure identifier */
    MQLONG   Version;          /* Structure version number */
    MQLONG   ExitOptions;      /* Reserved */
    MQLONG   AppOptions;       /* Application options */
    MQLONG   Encoding;         /* Numeric encoding required by
                               application */
    MQLONG   CodedCharSetId;    /* Character set required by application */
    MQLONG   DataLength;        /* Length in bytes of message data */
    MQLONG   CompCode;         /* Completion code */
    MQLONG   Reason;           /* Reason code qualifying CompCode */
    MQLONG   ExitResponse;      /* Response from exit */
    MQHCONN  Hconn;            /* Connection handle */
    PMQIEP   pEntryPoints;     /* Address of the MQIEP structure */
};
```

COBOL declaration (IBM i only)

```
**  MQDXP structure
10 MQDXP.
**  Structure identifier
15 MQDXP-STRUCID      PIC X(4).
**  Structure version number
15 MQDXP-VERSION      PIC S9(9) BINARY.
**  Reserved
15 MQDXP-EXITOPTIONS  PIC S9(9) BINARY.
**  Application options
15 MQDXP-APPOPTIONS   PIC S9(9) BINARY.
**  Numeric encoding required by application
```

```

15 MQDXP-ENCODING      PIC S9(9) BINARY.
** Character set required by application
15 MQDXP-CODEDCHARSETID PIC S9(9) BINARY.
** Length in bytes of message data
15 MQDXP-DATALENGTH    PIC S9(9) BINARY.
** Completion code
15 MQDXP-COMPCODE      PIC S9(9) BINARY.
** Reason code qualifying COMPCODE
15 MQDXP-REASON        PIC S9(9) BINARY.
** Response from exit
15 MQDXP-EXITRESPONSE  PIC S9(9) BINARY.
** Connection handle
15 MQDXP-HCONN         PIC S9(9) BINARY.

```

System/390 assembler declaration

```

MQDXP          DSECT
MQDXP_STRUCID  DS    CL4  Structure identifier
MQDXP_VERSION  DS    F    Structure version number
MQDXP_EXITOPTIONS DS    F    Reserved
MQDXP_APPOPTIONS DS    F    Application options
MQDXP_ENCODING DS    F    Numeric encoding required by application
MQDXP_CODEDCHARSETID DS    F    Character set required by application
MQDXP_DATALENGTH DS    F    Length in bytes of message data
MQDXP_COMPCODE DS    F    Completion code
MQDXP_REASON   DS    F    Reason code qualifying COMPCODE
MQDXP_EXITRESPONSE DS    F    Response from exit
MQDXP_HCONN    DS    F    Connection handle
*
MQDXP_LENGTH   EQU    *-MQDXP
               ORG    MQDXP
MQDXP_AREA     DS     CL(MQDXP_LENGTH)

```

MQXCNVC - Convert characters

The MQXCNVC call converts characters from one character set to another using the C programming language.

This call is part of the IBM MQ Data Conversion Interface (DCI), which is one of the IBM MQ framework interfaces.

Note: The call can be used from both application, and data-conversion exit environments.

Syntax

MQXCNVC (*Hconn*, *Options*, *SourceCCSID*, *SourceLength*, *SourceBuffer*, *TargetCCSID*, *TargetLength*, *TargetBuffer*, *DataLength*, *CompCode*, *Reason*)

Parameters

Hconn

Type: MQHCONN - input

This handle represents the connection to the queue manager.

In a data-conversion exit, *Hconn* is normally the handle that is passed to the data-conversion exit in the *Hconn* field of the MQDXP structure; this handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

On IBM i, the following special value can be specified for *Hconn*:

MQHC_DEF_HCONN

Default connection handle.

If you run a CICS TS 3.2 or higher application, ensure that the character conversion exit program, which invokes the MQXCNVC call, is defined as OPENAPI. This definition prevents the 2018 MQRC_HCONN_ERROR error caused by from an incorrect connection, and allows the MQGET to complete.

Options

Type: MQLONG - input

Options that control the action of MQXCNCV.

Zero or more of the options described can be specified. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

Default-conversion option: The following option controls the use of default character conversion:

MQDCC_DEFAULT_CONVERSION

Default conversion.

This option specifies that default character conversion can be used if one or both of the character sets specified on the call is not supported. This allows the queue manager to use an installation-specified default character set that approximates the specified character set, when converting the string.

Note: The result of using an approximate character set to convert the string is that some characters can be converted incorrectly. This can be avoided by using in the string only characters which are common to both the specified character set and the default character set.

The default character sets are defined by a configuration option when the queue manager is installed or restarted.

If MQDCC_DEFAULT_CONVERSION is not specified, the queue manager uses only the specified character sets to convert the string, and the call fails if one or both of the character sets is not supported.

This option is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows.

Padding option: The following option allows the queue manager to pad the converted string with blanks or discard insignificant trailing characters, in order to make the converted string fit the target buffer:

MQDCC_FILL_TARGET_BUFFER

Fill target buffer.

This option requests that conversion take place in such a way that the target buffer is filled completely:

- If the string contracts when it is converted, trailing blanks are added in order to fill the target buffer.
- If the string expands when it is converted, trailing characters that are not significant are discarded to make the converted string fit the target buffer. If this can be done successfully, the call completes with MQCC_OK and reason code MQRC_NONE.

If there are too few insignificant trailing characters, as much of the string as can fit is placed in the target buffer, and the call completes with MQCC_WARNING and reason code MQRC_CONVERTED_MSG_TOO_BIG.

Insignificant characters are:

- Trailing blanks
- Characters following the first null character in the string (but excluding the first null character itself)
- If the string, *TargetCCSID* , and *TargetLength* are such that the target buffer cannot be set completely with valid characters, the call fails with MQCC_FAILED and reason code MQRC_TARGET_LENGTH_ERROR. This can occur when *TargetCCSID* is a pure DBCS character set (such as UCS-2, but *TargetLength* specifies a length that is an odd number of bytes.
- *TargetLength* can be less than or greater than *SourceLength* . On return from MQXCNCV, *DataLength* has the same value as *TargetLength* .

If this option is not specified:

- The string is allowed to contract or expand within the target buffer as required. Insignificant trailing characters are not added or discarded.

If the converted string fits in the target buffer, the call completes with MQCC_OK and reason code MQRC_NONE.

If the converted string is too large for the target buffer, as much of the string as fits is placed in the target buffer, and the call completes with MQCC_WARNING and reason code MQRC_CONVERTED_MSG_TOO_BIG. Note fewer than *TargetLength* bytes can be returned in this case.

- *TargetLength* can be less than or greater than *SourceLength*. On return from MQXCNCV, *DataLength* is less than or equal to *TargetLength*.

This option is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Linux, Windows.

Encoding options: The options described can be used to specify the integer encodings of the source and target strings. The relevant encoding is used *only* when the corresponding character set identifier indicates that the representation of the character set in main storage is dependent on the encoding used for binary integers. This affects only certain multibyte character sets (for example, UCS-2 character sets).

The encoding is ignored if the character set is a single-byte character set (SBCS), or a multibyte character set with representation in main storage that is not dependent on the integer encoding.

Only one of the MQDCC_SOURCE_* values must be specified, combined with one of the MQDCC_TARGET_* values:

MQDCC_SOURCE_ENC_NATIVE

Source encoding is the default for the environment and programming language.

MQDCC_SOURCE_ENC_NORMAL

Source encoding is normal.

MQDCC_SOURCE_ENC_REVERSED

Source encoding is reversed.

MQDCC_SOURCE_ENC_UNDEFINED

Source encoding is undefined.

MQDCC_TARGET_ENC_NATIVE

Target encoding is the default for the environment and programming language.

MQDCC_TARGET_ENC_NORMAL

Target encoding is normal.

MQDCC_TARGET_ENC_REVERSED

Target encoding is reversed.

MQDCC_TARGET_ENC_UNDEFINED

Target encoding is undefined.

The encoding values defined previously can be added directly to the *Options* field. However, if the source or target encoding is obtained from the *Encoding* field in the MQMD or other structure, the following processing must be done:

1. The integer encoding must be extracted from the *Encoding* field by eliminating the float and packed-decimal encodings; see [“Analyzing encodings” on page 865](#) for details of how to do this.
2. The integer encoding resulting from step 1 must be multiplied by the appropriate factor before being added to the *Options* field. These factors are:
 - MQDCC_SOURCE_ENC_FACTOR for the source encoding
 - MQDCC_TARGET_ENC_FACTOR for the target encoding

The following example code illustrates how this might be coded in the C programming language:

```
Options = (MsgDesc.Encoding & MQENC_INTEGER_MASK)
          * MQDCC_SOURCE_ENC_FACTOR
          + (DataConvExitParms.Encoding & MQENC_INTEGER_MASK)
          * MQDCC_TARGET_ENC_FACTOR;
```

If not specified, the encoding options default to undefined (MQDCC_*_ENC_UNDEFINED). In most cases, this does not affect the successful completion of the MQXCNVC call. However, if the corresponding character set is a multibyte character set with representation that is dependent on the encoding (for example, a UCS-2 character set), the call fails with reason code MQRC_SOURCE_INTEGER_ENC_ERROR or MQRC_TARGET_INTEGER_ENC_ERROR as appropriate.

The encoding options are supported in the following environments: AIX, HP-UX, z/OS, IBM i, Solaris, Linux, Windows.

Default option: If none of the options described previously is specified, the following option can be used:

MQDCC_NONE

No options specified.

MQDCC_NONE is defined to aid program documentation. It is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

SourceCCSID

Type: MQLONG - input

This is the coded character set identifier of the input string in *SourceBuffer*.

SourceLength

Type: MQLONG - input

This is the length in bytes of the input string in *SourceBuffer* ; it must be zero or greater.

SourceBuffer

Type: MQCHARxSourceLength - input

This is the buffer containing the string to be converted from one character set to another.

TargetCCSID

Type: MQLONG - input

This is the coded character set identifier of the character set to which *SourceBuffer* is to be converted.

TargetLength

Type: MQLONG - input

This is the length in bytes of the output buffer *TargetBuffer* ; it must be zero or greater. It can be less than or greater than *SourceLength*.

TargetBuffer

Type: MQCHARxTargetLength - output

This is the string after it has been converted to the character set defined by *TargetCCSID*. The converted string can be shorter or longer than the unconverted string. The *DataLength* parameter indicates the number of valid bytes returned.

DataLength

Type: MQLONG - output

This is the length of the string returned in the output buffer *TargetBuffer*. The converted string can be shorter or longer than the unconverted string.

CompCode

Type: MQLONG - output

It is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CONVERTED_MSG_TOO_BIG

(2120, X'848') Converted data too large for buffer.

If *CompCode* is MQCC_FAILED:

MQRC_DATA_LENGTH_ERROR

(2010, X'7DA') Data length parameter not valid.

MQRC_DBCS_ERROR

(2150, X'866') DBCS string not valid.

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available.

MQRC_SOURCE_BUFFER_ERROR

(2145, X'861') Source buffer parameter not valid.

MQRC_SOURCE_CCSID_ERROR

(2111, X'83F') Source coded character set identifier not valid.

MQRC_SOURCE_INTEGER_ENC_ERROR

(2112, X'840') Source integer encoding not recognized.

MQRC_SOURCE_LENGTH_ERROR

(2143, X'85F') Source length parameter not valid.

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available.

MQRC_TARGET_BUFFER_ERROR

(2146, X'862') Target buffer parameter not valid.

MQRC_TARGET_CCSID_ERROR

(2115, X'843') Target coded character set identifier not valid.

MQRC_TARGET_INTEGER_ENC_ERROR

(2116, X'844') Target integer encoding not recognized.

MQRC_TARGET_LENGTH_ERROR

(2144, X'860') Target length parameter not valid.

MQRC_UNEXPECTED_ERROR

(2195, X'893') Unexpected error occurred.

For detailed information about these codes, see [Messages and reason codes](#).

C invocation

```
MQXCNCV (Hconn, Options, SourceCCSID, SourceLength, SourceBuffer,  
         TargetCCSID, TargetLength, TargetBuffer, &DataLength,  
         &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;           /* Connection handle */  
MQLONG   Options;         /* Options that control the action of  
                           MQXCNCV */  
MQLONG   SourceCCSID;     /* Coded character set identifier of string  
                           before conversion */  
MQLONG   SourceLength;    /* Length of string before conversion */  
MQCHAR   SourceBuffer[n]; /* String to be converted */  
MQLONG   TargetCCSID;     /* Coded character set identifier of string  
                           after conversion */  
MQLONG   TargetLength;    /* Length of output buffer */  
MQCHAR   TargetBuffer[n]; /* String after conversion */  
MQLONG   DataLength;      /* Length of output string */  
MQLONG   CompCode;        /* Completion code */  
MQLONG   Reason;          /* Reason code qualifying CompCode */
```

COBOL declaration (IBM i only)

```
CALL 'MQXCNCV' USING HCONN, OPTIONS, SOURCECCSID, SOURCELENGTH,  
                     SOURCEBUFFER, TARGETCCSID, TARGETLENGTH,  
                     TARGETBUFFER, DATALENGTH, COMPCODE, REASON.
```

Declare the parameters as follows:

```
** Connection handle  
01 HCONN          PIC S9(9) BINARY.  
** Options that control the action of MQXCNCV  
01 OPTIONS        PIC S9(9) BINARY.  
** Coded character set identifier of string before conversion  
01 SOURCECCSID    PIC S9(9) BINARY.  
** Length of string before conversion  
01 SOURCELENGTH   PIC S9(9) BINARY.  
** String to be converted  
01 SOURCEBUFFER    PIC X(n).  
** Coded character set identifier of string after conversion  
01 TARGETCCSID    PIC S9(9) BINARY.  
** Length of output buffer  
01 TARGETLENGTH   PIC S9(9) BINARY.  
** String after conversion  
01 TARGETBUFFER    PIC X(n).  
** Length of output string  
01 DATALENGTH    PIC S9(9) BINARY.  
** Completion code  
01 COMPCODE        PIC S9(9) BINARY.  
** Reason code qualifying COMPCODE  
01 REASON          PIC S9(9) BINARY.
```

S/390 assembler declaration

```
CALL MQXCNCV, (HCONN, OPTIONS, SOURCECCSID, SOURCELENGTH,      X  
               SOURCEBUFFER, TARGETCCSID, TARGETLENGTH, TARGETBUFFER, X  
               DATALENGTH, COMPCODE, REASON)
```

Declare the parameters as follows:

```
HCONN      DS F      Connection handle  
OPTIONS     DS F      Options that control the action of MQXCNCV  
SOURCECCSID DS F      Coded character set identifier of string before  
* conversion  
SOURCELENGTH DS F      Length of string before conversion  
SOURCEBUFFER DS CL(n) String to be converted  
TARGETCCSID DS F      Coded character set identifier of string after  
* conversion  
TARGETLENGTH DS F      Length of output buffer  
TARGETBUFFER DS CL(n) String after conversion
```

DATALENGTH	DS	F	Length of output string
COMPCODE	DS	F	Completion code
REASON	DS	F	Reason code qualifying COMPCODE

MQ_DATA_CONV_EXIT - Data conversion exit

The MQ_DATA_CONV_EXIT call describes the parameters that are passed to the data-conversion exit.

No entry point called MQ_DATA_CONV_EXIT is provided by the queue manager (see usage note [11](#)).

This definition is part of the IBM MQ Data Conversion Interface (DCI), which is one of the IBM MQ framework interfaces.

Syntax

MQ_DATA_CONV_EXIT (*DataConvExitParms*, *MsgDesc*, *InBufferLength*, *InBuffer*, *OutBufferLength*, *OutBuffer*)

Parameters

DataConvExitParms

Type: MQDXP - input/output

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate the outcome of the conversion. See [“MQDXP - Data-conversion exit parameter”](#) on page 876 for details of the fields in this structure.

MsgDesc

Type: MQMD - input/output

On input to the exit, this is the message descriptor associated with the message data passed to the exit in the *InBuffer* parameter.

Note: The *MsgDesc* parameter passed to the exit is always the most-recent version of MQMD supported by the queue manager which invokes the exit. If the exit is intended to be portable between different environments, the exit will check the *Version* field in *MsgDesc* to verify that the fields that the exit needs to access are present in the structure.

In the following environments, the exit is passed a version-2 MQMD: AIX, HP-UX, IBM i, Solaris, Linux, Windows. In all other environments that support the data conversion exit, the exit is passed a version-1 MQMD.

On output, the exit will change the *Encoding* and *CodedCharSetId* fields to the values requested by the application, if conversion was successful; these changes are reflected back to the application. Any other changes that the exit makes to the structure are ignored; they are not reflected back to the application.

If the exit returns MQXDR_OK in the *ExitResponse* field of the MQDXP structure, but does not change the *Encoding* or *CodedCharSetId* fields in the message descriptor, the queue manager returns for those fields the values that the corresponding fields in the MQDXP structure had on input to the exit.

InBufferLength

Type: MQLONG - input

Length in bytes of *InBuffer*.

This is the length of the input buffer *InBuffer*, and specifies the number of bytes to be processed by the exit. *InBufferLength* is the lesser of the length of the message data before conversion, and the length of the buffer provided by the application on the MQGET call.

The value is always greater than zero.

InBuffer

Type: MQBYTEInBufferLength - input

Buffer containing the unconverted message.

This contains the message data before conversion. If the exit is unable to convert the data, the queue manager returns the contents of this buffer to the application after the exit has completed.

Note: The exit should not alter *InBuffer* ; if this parameter is altered, the results are undefined.

In the C programming language, this parameter is defined as a pointer-to-void.

OutBufferLength

Type: MQLONG - input

Length in bytes of *OutBuffer*.

This is the length of the output buffer *OutBuffer*, and is the same as the length of the buffer provided by the application on the MQGET call.

The value is always greater than zero.

OutBuffer

Type: MQBYTEOutBufferLength - output

Buffer containing the converted message.

On output from the exit, if the conversion was successful (as indicated by the value MQXDR_OK in the *ExitResponse* field of the *DataConvExitParms* parameter), *OutBuffer* contains the message data to be delivered to the application, in the requested representation. If the conversion was unsuccessful, any changes that the exit has made to this buffer are ignored.

In the C programming language, this parameter is defined as a pointer-to-void.

Usage notes

1. A data-conversion exit is a user-written exit which receives control during the processing of an MQGET call. The function performed by the data-conversion exit is defined by the provider of the exit; however, the exit must conform to the rules described here, and in the associated parameter structure MQDXP.

The programming languages that can be used for a data-conversion exit are determined by the environment.

2. The exit is invoked only if *all* of the following are true:

- The MQGMO_CONVERT option is specified on the MQGET call
- The *Format* field in the message descriptor is not MQFMT_NONE
- The message is not already in the required representation; that is, one or both of the message's *CodedCharSetId* and *Encoding* is different from the value specified by the application in the message descriptor supplied on the MQGET call
- The queue manager has not already done the conversion successfully
- The length of the application's buffer is greater than zero
- The length of the message data is greater than zero
- The reason code so far during the MQGET operation is MQRC_NONE or MQRC_TRUNCATED_MSG_ACCEPTED

3. When an exit is being written, consider coding the exit in a way that allows it to convert messages that have been truncated. Truncated messages can arise in the following ways:

- The receiving application provides a buffer that is smaller than the message, but specifies the MQGMO_ACCEPT_TRUNCATED_MSG option on the MQGET call.

In this case, the *Reason* field in the *DataConvExitParms* parameter on input to the exit has the value MQRC_TRUNCATED_MSG_ACCEPTED.

- The sender of the message truncated it before sending it. This can happen with report messages, for example (see [“Conversion of report messages”](#) on page 876 for more details).

In this case, the *Reason* field in the *DataConvExitParms* parameter on input to the exit has the value MQRC_NONE (if the receiving application provided a buffer that was large enough for the message).

Thus the value of the *Reason* field on input to the exit cannot always be used to decide whether the message has been truncated.

The distinguishing characteristic of a truncated message is that the length provided to the exit in the *InBufferLength* parameter is *less than* the length implied by the format name contained in the *Format* field in the message descriptor. The exit should therefore check the value of *InBufferLength* before attempting to convert any of the data; the exit *should not* assume that the full amount of data implied by the format name has been provided.

If the exit has *not* been written to convert truncated messages, and *InBufferLength* is less than the value expected, the exit will return MQXDR_CONVERSION_FAILED in the *ExitResponse* field of the *DataConvExitParms* parameter, with the *CompCode* and *Reason* fields set to MQCC_WARNING and MQRC_FORMAT_ERROR.

If the exit *has* been written to convert truncated messages, the exit will convert as much of the data as possible (see next usage note), taking care not to attempt to examine or convert data beyond the end of *InBuffer*. If the conversion completes successfully, the exit will leave the *Reason* field in the *DataConvExitParms* parameter unchanged. This returns MQRC_TRUNCATED_MSG_ACCEPTED if the message was truncated by the receiver's queue manager, and MQRC_NONE if the message was truncated by the sender of the message.

It is also possible for a message to expand *during* conversion, to the point where it is bigger than *OutBuffer*. In this case the exit must decide whether to truncate the message; the *AppOptions* field in the *DataConvExitParms* parameter indicates whether the receiving application specified the MQGMO_ACCEPT_TRUNCATED_MSG option.

4. Generally, all the data in the message provided to the exit in *InBuffer* is converted, or that none of it is. An exception to this, however, occurs if the message is truncated, either before conversion or during conversion; in this case there can be an incomplete item at the end of the buffer (for example: 1 byte of a double-byte character, or 3 bytes of a 4-byte integer). In this situation, consider omitting the incomplete item and set the unused bytes in the *OutBuffer* to nulls. However, complete elements or characters within an array or string *should* be converted.
5. When an exit is needed for the first time, the queue manager attempts to load an object that has the same name as the format (apart from extensions). The object loaded must contain the exit that processes messages with that format name. Consider making the exit name, and the name of the object that contains the exit identical, although not all environments require this.
6. A new copy of the exit is loaded when an application attempts to retrieve the first message that uses that *Format* since the application connected to the queue manager. For CICS or IMS applications, this means when the CICS or IMS subsystem connected to the queue manager. A new copy can also be loaded at other times, if the queue manager has discarded a previously loaded copy. For this reason, an exit must not attempt to use static storage to communicate information from one invocation of the exit to the next - the exit can be unloaded between the two invocations.
7. If there is a user-supplied exit with the same name as one of the built-in formats supported by the queue manager, the user-supplied exit does not replace the built-in conversion routine. The only circumstances in which such an exit is invoked are:
 - If the built-in conversion routine cannot handle conversions to or from either the *CodedCharSetId* or *Encoding* involved, or
 - If the built-in conversion routine has failed to convert the data (for example, because there is a field or character which cannot be converted).
8. The scope of the exit is environment-dependent. *Format* names must be chosen to minimize the risk of clashes with other formats. Consider starting with characters that identify the application defining the format name.

9. The data-conversion exit runs in an environment like that of the program which issued the MQGET call; environment includes address space and user profile (where applicable). The program could be a message channel agent sending messages to a destination queue manager that does not support message conversion. The exit cannot compromise the queue manager's integrity, since it does not run in the queue manager's environment.
10. The only MQI call which can be used by the exit is MQXCNVC; attempting to use other MQI calls fails with reason code MQRC_CALL_IN_PROGRESS, or other unpredictable errors.
11. No entry point called MQ_DATA_CONV_EXIT is provided by the queue manager. However, a typedef is provided for the name MQ_DATA_CONV_EXIT in the C programming language, and this can be used to declare the user-written exit, to ensure that the parameters are correct. The name of the exit must be the same as the format name (the name contained in the *Format* field in MQMD), although this is not required in all environments.

The following example illustrates how the exit that processes the format MYFORMAT can be declared in the C programming language:

```
#include "cmqc.h"
#include "cmqxc.h"

MQ_DATA_CONV_EXIT MYFORMAT;

void MQENTRY MYFORMAT(
    PMQDXP  pDataConvExitParms, /* Data-conversion exit parameter
                                block */
    PMQMD   pMsgDesc,           /* Message descriptor */
    MQLONG  InBufferLength,     /* Length in bytes of InBuffer */
    PMQVOID pInBuffer,          /* Buffer containing the unconverted
                                message */
    MQLONG  OutBufferLength,    /* Length in bytes of OutBuffer */
    PMQVOID pOutBuffer)         /* Buffer containing the converted
                                message */
{
    /* C language statements to convert message */
}
```

12. On z/OS, if an API-crossing exit is also in force, it is called after the data-conversion exit.

C invocation

```
exitname (&DataConvExitParms, &MsgDesc, InBufferLength,
          InBuffer, OutBufferLength, OutBuffer);
```

The parameters passed to the exit are declared as follows:

```
MQDXP  DataConvExitParms; /* Data-conversion exit parameter block */
MQMD   MsgDesc;           /* Message descriptor */
MQLONG InBufferLength;    /* Length in bytes of InBuffer */
MQBYTE InBuffer[n];       /* Buffer containing the unconverted
                             message */
MQLONG OutBufferLength;   /* Length in bytes of OutBuffer */
MQBYTE OutBuffer[n];      /* Buffer containing the converted
                             message */
```

COBOL declaration (IBM i only)

```
CALL 'exitname' USING DATACONVEXITPARMS, MSGDESC, INBUFFERLENGTH,
                     INBUFFER, OUTBUFFERLENGTH, OUTBUFFER.
```

The parameters passed to the exit are declared as follows:

```
** Data-conversion exit parameter block
01 DATACONVEXITPARMS.
   COPY CMQDXPV.
** Message descriptor
01 MSGDESC.
   COPY CMQMDV.
** Length in bytes of INBUFFER
01 INBUFFERLENGTH PIC S9(9) BINARY.
```

```

** Buffer containing the unconverted message
01 INBUFFER          PIC X(n).
** Length in bytes of OUTBUFFER
01 OUTBUFFERLENGTH   PIC S9(9) BINARY.
** Buffer containing the converted message
01 OUTBUFFER          PIC X(n).

```

System/390 assembler declaration

```

CALL EXITNAME, (DATACONVEXITPARMS, MSGDESC, INBUFFERLENGTH,      X
                INBUFFER, OUTBUFFERLENGTH, OUTBUFFER)

```

The parameters passed to the exit are declared as follows:

```

DATACONVEXITPARMS  CMQDXPA  ,      Data-conversion exit parameter block
MSGDESC            CMQMDA   ,      Message descriptor
INBUFFERLENGTH     DS       F      Length in bytes of INBUFFER
INBUFFER           DS       CL(n)  Buffer containing the unconverted
*                  message
OUTBUFFERLENGTH     DS       F      Length in bytes of OUTBUFFER
OUTBUFFER          DS       CL(n)  Buffer containing the converted
*                  message

```

Properties specified as MQRFH2 elements

Non-message descriptor properties can be specified as elements in MQRFH2 header folders. Overview of MQRFH2 elements being specified as properties.

This retains compatibility with the previous versions of the IBM MQ JMS and XMS clients. This section describes how to specify properties in MQRFH2 headers.

To use MQRFH2 elements as properties, specify the elements as described in [Using IBM MQ classes for Java](#). This information supplements the information described in “MQRFH2 - Rules and formatting header 2” on page 502.

Mapping property data types to MQRFH2 data types

This topic provides information on message property types mapped to their corresponding MQRFH2 data types.

Table 568. Supported MQRFH2 data types	
Message property type	MQRFH2 data type
MQBYTE[]	bin.hex
MQBOOL	boolean
MQINT8	i1
MQINT16	i2
MQINT32	i4
MQINT64	i8
MQFLOAT32	r4
MQFLOAT64	r8
MQCHAR[]	string

Any element without a data type is assumed to be of type "string".

An MQRFH2 data type of `int`, meaning an integer of unspecified size, is treated as if it were an `i8`.

A null value is indicated by the element attribute `xsi:nil='true'`. Do not use the attribute `xsi:nil='false'` for non-null values.

For example, the following property has a null value:

```
<NullProperty xsi:nil='true'></NullProperty>
```

A byte or character string property can have an empty value. This is represented by an MQRFH2 element with a zero length element value.

For example, the following property has an empty value:

```
<EmptyProperty></EmptyProperty>
```

Supported MQRFH2 folders

Overview of the use of message descriptor fields as properties.

The folders `<jms>`, `<mcd>`, `<mqext>`, and `<usr>` are described in [The MQRFH2 header and JMS](#). The `<usr>` folder is used to transport any JMS application-defined properties that are associated with a message. Groups are not allowed in the `<usr>` folder.

[The MQRFH2 header and JMS](#) supports the following additional folders:

- `<mq>`

This folder is used and reserved for MQ-defined properties that are used by IBM MQ.

- `<mq_usr>`

This folder can be used to transport any application-defined properties that are not exposed as JMS user-defined properties, as the properties might not meet the requirements of a JMS property. This folder can contain groups that the `<usr>` folder cannot.

- Any folder marked with the `content='properties'` attribute.

Such a folder is equivalent to the `<mq_usr>` folder in content.

- `<mpps>`

This folder is used for IBM MQ publish/subscribe properties.

IBM MQ also supports the following folders that are already in use by WAS/SIB:

- `<sib>`

This folder is used and reserved for WAS/SIB system message properties that are not exposed as JMS properties, or are mapped to `JMS_IBM_*` properties, but are exposed to WAS/SIB applications; these include forward and reverse routing paths properties.

At least some cannot be exposed as JMS properties, because they are byte arrays. If your application adds properties to this folder, the value is either ignored or removed.

- `<sib_usr>`

This folder is used and reserved for WAS/SIB user message properties that cannot be exposed as JMS user properties because they are not of supported types; they are exposed to WAS/SIB applications.

These are user properties, that you can get or set through the `SIMessage` interface, but the content of the byte array is mapped to the required property value.

If your IBM MQ application writes an arbitrary `bin.hex` element to the folder, the application probably receives an `IOException`, as it is not of the format expected to restore. If you add anything other than a `bin.hex` element you receive a `ClassCastException`.

Do not attempt to make properties available to WAS/SIB by using this folder; instead user the `<usr>` folder for that purpose.

- `<sib_context>`

This folder is used for WAS/SIB system message properties that are not exposed to WAS/SIB user applications or as JMS properties. These include security and transactional properties that are used for web services and similar.

Your application must not add properties to this folder.

- <mqema>

This folder was used by WAS/SIB instead of the <mqext> folder.

MQRFH2 folder names are case-sensitive.

The following folders are reserved, in any mixture of lowercase or uppercase characters:

- Any folder prefixed by mq or wmq ; reserved for use by IBM MQ.
- Any folder prefixed by sib ; reserved for use by WAS/SIB.
- <Root> and <Body> folders; reserved but not used.

The following folders are not recognized as containing message properties:

- <psc>

Used by IBM Integration Bus to convey publish/subscribe command messages to the broker.

- <pscr>

Used by IBM Integration Bus to contain information from the broker, in response to publish/subscribe command messages.

- Any folder not defined by IBM, that is not marked with the content= 'properties' attribute.

Do not specify content= 'properties' on the <psc> or <pscr> folders. If you do so, these folders are treated as properties and IBM Integration Bus is likely to stop functioning as expected.

If your application is building messages with properties, in MQRFH2 headers to be recognized as an MQRFH2 header containing properties, the header must be in the list of headers that can be chained at the head of the message.

The MQRFH2 can be preceded by any number of MQH standard headers, or an MQCIH, an MQDLH, an MQIIH, an MQTM, an MQTMC2, or an MQXQH. A string or an MQCFH ends parsing because they cannot be chained.

It is possible for a message to contain multiple MQRFH2 headers all carrying message properties. Folders with the same name can coexist in different headers unless otherwise restricted, for example by WAS/SIB. The folders are treated as one logical folder, if they are all in significant headers.

While folders from the significant headers cannot be merged with those folders in nonsignificant headers, folders with the same name within the significant headers can be merged, removing any conflicting properties. Your applications must not depend on the layout of properties within their message.

MQRFH2 groups are parsed for properties in user-defined folders, that is, not the <wmq>, <jms>, <mcd>, <usr>, <mqext>, <sib>, <sib_usr>, <sib_context>, and <mqema> folders.

Groups in the IBM-defined properties folders, except for the <wmq> and <mq> folders, are parsed for properties.

An MQRFH2 folder cannot contain mixed content; a folder or group can contain either groups or properties, or a value, but not both.

A segment of a message, either the first or a subsequent segment, cannot contain IBM MQ -defined properties other than those properties in the message descriptor. Therefore putting a message containing such properties with either MQMF_SEGMENT or MQMF_SEGMENTATION_ALLOWED set causes the put to fail with MQRC_SEGMENTATION_NOT_ALLOWED.

However, message groups can contain IBM MQ -defined properties.


Generation of MQRFH2 headers

If IBM MQ converts message properties to their MQRFH2 representation, it must add the MQRFH2 to the message. It adds the MQRFH2 either as a separate header, or merges it with an existing header.

Generation of new MQRFH2 headers by IBM MQ might disrupt existing headers in a message. Applications that parse a message buffer for headers must be aware that the number and position of headers in

a buffer might change in some circumstances. IBM MQ attempts to minimize the impact of adding properties to a message by merging message properties into an existing MQRFH2 header, where it can. It also attempts to minimize the impact by inserting a generated MQRFH2 into a fixed position relative to other headers in the message buffer.

A generated MQRFH2 header is placed following the MQMD, and any number of MQXQH, MQRFH, and MQDLH headers, whatever order they are in. The generated MQRFH2 header is placed immediately before the first header that is not an MQMD, MQXQH, MQDLH, or MQRFH header.

 On z/OS systems, the generated MQRFH2 header is created in the application's CCSID. This is defined as follows:

- For batch LE applications using the DLL interface, the CCSID is the CODESET associated with the current locale at the time **MQCONN** is issued (default value is 1047).
- For batch LE applications bound with one of the batch MQ stubs, the CCSID is the CODESET associated with the current locale at the time of the first MQI call issued after **MQCONN** (default value is 1047).
- For batch non-LE applications running on a USS thread, the CCSID is the value of THLICCSID at the time of the first MQI call issued after **MQCONN** (default value is 1047).
- For other batch applications, the CCSID is the CCSID of the queue manager.

For LE applications, the locale can be changed using the `setlocale()` / `CEESETL` LE callable service. For non-LE applications running on USS threads, the value of THLICCSID can be changed using USS mapping macro **BPXYTHLI**.

Rules for merging generated MQRFH2

The following rules apply to merging a generated MQRFH2 with an existing MQRFH2. The generated MQRFH2 header is merged with an existing MQRFH2 header, if:

1. The existing MQRFH2 is in the same position IBM MQ would place a generated MQRFH2, or earlier in the header chain.
2. The CCSID of the generated properties is the same as the NameValueCCSID of the existing MQRFH2.

Otherwise, the generated header is placed separately in the buffer, in the position described before.

Rules for merging folders in an existing MQRFH2

If message properties are merged into an existing MQRFH2, then the existing MQRFH2 is scanned for folders that match the message properties, and merges them. If a matching folder does not exist a new folder is added to the end of the existing folders. If a matching folder does exist, the folder is searched. Any matching properties are overwritten. Any new ones are added at the end of the folder.

MQRFH2 folder restrictions

Overview of folder restrictions in MQRFH2 headers

The MQRFH2 restrictions apply to the following folders:

- Element names in the `<usr>` folder must not begin with the prefix JMS ; such property names are reserved for use by JMS and are not valid for user-defined properties.

Such an element name does not cause parsing of the MQRFH2 to fail, but is not accessible to the IBM MQ message property APIs.

- Element names in the `<usr>` folder must not be, in any mixture of lower or uppercase, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, IS and ESCAPE. These names match SQL keywords and make parsing selectors harder, because `<usr>` is the default folder used when no folder is specified for a particular property in a selector.

Such an element name does not cause parsing of the MQRFH2 to fail, but is not accessible to the IBM MQ message property APIs.

- The content model of the <usr> folder is as follows:
 - Any valid XML name can be used as an element name, providing that it does not contain a colon.
 - Only simple elements, not nested folders, are allowed.
 - All elements take the default type of string, unless modified by a dt="xxx" attribute.
 - All elements are optional, but should occur no more than once in a folder.
- Element names in any folder considered to contain message properties must not contain a period (.) (Unicode character U+002E), because this is used in property names to indicate the hierarchy.

Such an element name does not cause parsing of the MQRFH2 to fail, but is not accessible to the IBM MQ message property APIs.

In general, MQRFH2 headers that contain valid XML-style data can be parsed by IBM MQ without failure, although certain elements of the MQRFH2 are not accessible through the IBM MQ message property APIs.

MQRFH2 element name conflicts

Overview of conflicts within MQRFH2 element names.

Only one value can be attached to a message property. If an attempt to access a property leads to a conflict of values, one is chosen in preference over another.

The IBM MQ syntax for accessing MQRFH2 elements allows unique identification of an element, if a folder contains no elements with the same name. If a folder contains more than one element with the same name, the value of the property used is the one closest to the head of the message.

This applies if two or more folders of the same name are contained in different significant MQRFH2 headers within the same message.

A conflict can result when the MQGET call is processed after a non-message descriptor property has been set twice: both through an MQSETMP call and directly in the raw MQRFH2 header.

If this happens, the property associated with the message by an API call takes preference over one in the message data, that is, the one in the raw MQRFH2 header. If a conflict occurs, it is considered to come logically before the message data.

Mapping from property names to MQRFH2 folder and element names

Overview of the differences between property names and element names in the MQRFH2 header.

When using any of the defined APIs that ultimately generate MQRFH2 headers, in order to specify message properties (for example, MQ JMS), the property name is not necessarily the element name in the MQRFH2 folder.

Therefore, a mapping occurs from the property name to the MQRFH2 element, and in the reverse way, taking into account both the folder name that contains the element, and the element name. Some examples from IBM MQ classes for JMS are already documented in [Using Java](#).

Property name	MQRFH2 folder name	MQRFH2 element name
JMSDestination	jms	Dst
JMSType	mcd	Type, Set, Fmt
xxx (user defined, where xxx does not begin with JMS)	usr	xxx

Therefore, when a JMS application accesses the JMSDestination property this maps to the Dst element in the <jms> folder.

When specifying properties as MQRFH2 elements, IBM MQ defines its elements as follows:

Property name	MQRFH2 folder name	MQRFH2 group name	MQRFH2 element name
<Property>	<usr>	n/a	<Property>
<folder>.<Property>	<folder>	n/a	<Property>
<folder>.<group>.<Property>	<folder>	<group>	<Property>

For example, when an IBM MQ application attempts to access the Property1 property, this maps to the Property1 element in the <usr> folder. The wmq.Property2 property maps to the Property2 property in the <wmq> folder.

If the property name contains more than one . character, the MQRFH2 element name used is the one following the final . character, and MQRFH2 groups are used to form a hierarchy; nested MQRFH2 groups are permitted.

The JMS header and provider-specific properties that are contained in an MQRFH2 in the <mcd>, <jms>, and <mqext> folders are accessed by an IBM MQ application using the short names defined in [Using IBM MQ classes for Java](#).

JMS user-defined properties are accessed from the <usr> folder. An IBM MQ application can use the <usr> folder for its application properties if it is acceptable for the property to appear to JMS applications as one of its user-defined properties.

If it is not acceptable, choose another folder; the <wmq_usr> folder is provided as a standard location for such non-JMS properties.

Your applications can specify and use any MQRFH2 folder with a well-defined use, not documented in “Properties specified as MQRFH2 elements” on page 892 if you note the following:

1. The folder might already be in use, or might be used in the future, by another application providing undefined access to properties contained inside it; see [Property names](#) for the suggested naming convention for property names.
2. The properties are not accessible to previous versions of the IBM MQ classes for JMS or XMS client that can only access the <usr> folder for user-defined properties
3. The folder must be marked with the attribute content with the value set to properties, for example, content='properties'.

“MQSETMP - Set message property” on page 750 automatically adds this attribute as required. This attribute must not be added to any of the IBM-defined folders, for example, <jms> and <usr>. Doing so, causes the message to be rejected by the IBM MQ classes for JMS client before Version 7.0. with a MessageFormatException.

Because the <usr> folder is the default location for properties of the <Property> syntax, an IBM MQ application and a JMS application to access the same user-defined property value using the same name.

Reserved folder names

There are several reserved folder names. You cannot use such names as your folder prefixes; for example, Root.Property1 does not access a valid property because Root is reserved. The following list contains reserved folder names:

- Root
- Body
- Properties
- Environment
- LocalEnvironment
- DestinationList
- ExceptionList

- InputBody
- InputRoot
- InputProperties
- InputLocalEnvironment
- InputDestinationList
- InputExceptionList
- OutputRoot
- OutputLocalEnvironment
- OutputDestinationList
- OutputExceptionList

Mapping property descriptor fields into MQRFH2 headers

When a property is translated into an MQRFH2 element the following element attributes are used to specify the significant fields of the property descriptor: This describes how MQPD fields are translated to MQRFH2 element attributes.

Support

The Support property descriptor field is split into three element attributes

- The **sr** element attribute specifies values in the MQPD_REJECT_UNSUP_MASK bit mask.
- The **sa** element attribute specifies values in the MQPD_ACCEPT_UNSUP_MASK bit mask.
- The **sx** element attribute specifies values in the MQPD_ACCEPT_UNSUP_IF_XMIT_MASK bit mask.

These element attributes are only valid in the <mq> folder and are ignored if set on elements in the other folders containing properties.

Support value	MQRFH2 element attribute	MQRFH2 attribute value
MQPD_SUPPORT_OPTIONAL	sa	optional This is the default value.
MQPD_SUPPORT_REQUIRED	sr	required
MQPD_SUPPORT_REQUIRED_IF_LOCAL	sx	local

Context

Use the **context** element attribute to indicate the message context to which a property belongs. Use one value only. This element attribute is valid on a property in any folder containing properties.

Context value	MQRFH2 attribute value
MQPD_NO_CONTEXT	none This is the default value.
MQPD_USER_CONTEXT	user

CopyOptions

Use the **copy** element attribute to indicate messages into which a property should be copied. More than one value is acceptable; separate multiple values with a comma. For example **copy='reply'** and **copy='publish,report'** are both valid. This element attribute is valid on a property in any folder containing properties.

Note: In the attribute definition, single quotation marks or double quotation marks are valid use, for example **copy='reply'** or **copy="report"**

CopyOption value	MQRFH2 attribute value
MQPD_COPY_FORWARD	forward
MQPD_COPY_REPLY	reply
MQPD_COPY_REPORT	report
MQPD_COPY_PUBLISH	publish
MQPD_COPY_ALL	all Do not specify this with any other value. When used with another value, this takes precedence over any value except none .
MQPD_COPY_DEFAULT	default This is the default value. It is equivalent to specifying the three values MQCOPY_FORWARD, MQCOPY_REPORT and MQCOPY_PUBLISH. Do not specify this with any other value.
MQPD_COPY_NONE	none Do not specify this with any other value. When used with another value, this takes precedence.

Restrictions to the <mq> MQRFH2 folder

When a message is put on to a queue, it is searched for an <mq> folder so that the message can be processed according to its MQ-defined properties. To allow the efficient parsing of MQ-defined properties, the following restrictions apply to the folder:

- Only properties in the first significant <mq> folder in the message are acted upon by MQ; properties in any other <mq> folder in the message are ignored.
- If the folder is in UTF-8, only single-byte UTF-8 characters are allowed in the folder. A multi-byte character in the folder, can cause parsing to fail, and the message to be rejected.
- Do not include MQRFH2 groups in the <mq> folder. The presence of Unicode character U+003C in a property value will cause the message to be rejected.
- Do not use escape strings in the folder. An escape string is treated as the actual value of the element.
- Only Unicode character U+0020 is treated as white space within the folder. All other characters are treated as significant and can cause parsing of the folder to fail, and the message to be rejected.

If parsing of the <mq> folder fails, or if the folder does not observe these restrictions, the message is rejected with CompCode **MQCC_FAILED** and Reason **MQRC_RFH_RESTRICTED_FORMAT_ERR**.

MQRFH2 headers that are not valid

At the time an MQPUT, MQPUT1, or MQGET call processes, a partial parsing of any MQRFH2 headers in the message can occur to check what folders are included, and to determine if the folders contain properties. Overview of MQRFH2 headers that are not valid.

If the partial parsing of the message cannot complete successfully because the structure is not valid, for example, the StrucLength field is too small, then:

- The MQPUT or MQPUT1 call fails with reason code MQRC_RFH_ERROR, if it can be determined that the application includes some IBM MQ Version 7 option, so that existing applications do not fail.

- The MQGET call returns successfully, and the MQRFH2 containing the error is returned in the buffer you provided.

If the partial parsing fails because it cannot be detected whether a particular folder contains properties or not, for example, the folder begins <<jms, so parsing fails before the folder name is determined, then:

- The MQPUT or MQPUT1 call fails with reason code MQRC_RFH_FORMAT_ERROR, if it can be determined that the application includes some IBM MQ Version 7 option, so that existing applications do not fail.
- The MQGET call returns successfully, and the MQRFH2 containing the error is returned in the buffer you provided.
- While internally within the queue manager, the message is not rejected due to the badly formatted folder, but the folder is always treated as if no properties were contained inside it.

A message can flow through the queue manager network with a folder containing such a syntax error, but never being parsed and detected, while one or more folders in the message are:

- Valid
- Successfully parsed
- Used in the processing of the message

Therefore, detection is not guaranteed.

If one of your applications uses [“MQSETMP - Set message property”](#) on page 750, or MQINQMP to access a property, and in so doing this causes an MQRFH2 folder to be fully parsed, detecting an error such that parsing cannot complete, this is indicated by an appropriate return code to the API call. No properties in the folder are made available to the application.

If an attempt is made to fully parse an MQRFH2 folder and the parser finds unrecognized element attributes, or an unrecognized data type, parsing continues and complete successfully with no warnings being issued; this does not constitute a parsing error.

Code page conversion

This section describes codeset names and CCSIDs, national language, z/OS conversion, IBM i conversion, and Unicode conversion support.

Each national language section lists the following information:

- The native CCSIDs supported
- The code page conversions that are **not** supported

The following terms are used in the information:

-8

Indicates for HP-UX that the CCSID is for the HP-UX defined codeset *roman8*

AIX

Indicates IBM MQ for AIX

HP-UX

Indicates IBM MQ for HP-UX

Linux

Indicates IBM MQ for Linux for Intel and IBM MQ for Linux for zSeries

HP Integrity NonStop Server

Indicates IBM WebSphere MQ for HP Integrity NonStop Server

OS/400

Indicates IBM MQ for IBM i

Solaris

Indicates IBM MQ for Solaris

Windows

Indicates IBM MQ for Windows

z/OS

Indicates IBM MQ for z/OS

The default for data conversion is for the conversion to be performed at the target (receiving) system.

If the source product supports the conversion a channel can be set up and data exchanged by setting the channel attribute CONVERT to YES at the source.

Note:

1. Conversion for IBM MQ MQI client information takes place in the server, so the server must support conversion from the client CCSID to the server CCSID.
2. The conversion might include support added by CSD/PTF to the latest version of IBM MQ. Check the content of the latest service level to see if you need to install a CSD/PTF to enable this conversion.
3. The IBM MQ queue manager CCSID must be Mixed or SBCS.
4. Some CCSIDs, for example 850 on AIX, that are not supported by the operating system can still be used by the application and also can be set as the IBM MQ queue manager CCSID. This is allowed for backward compatibility purpose only and the conversion will fail if the relevant conversion tables are not installed.


See [Table 569 on page 901](#) for a cross reference between some of the CCSID numbers and some industry codeset names.

Codeset names and CCSIDs

IBM MQ for z/OS provides more conversion than is listed in the language specific tables.

<i>Table 569. Codeset names and CCSIDs</i>	
Codeset names	CCSIDs
ISO 8859-1	819
ISO 8859-2	912
ISO 8859-3	913
ISO 8859-5	915
ISO 8859-6	1089
ISO 8859-7	813
ISO 8859-8	916
ISO 8859-9	920
ISO 8859-13	921
ISO 8859-15 (euro)	923
big5	950
eucJP	954 5050 33722
eucKR	970
eucTW	964
eucCN	1383
PCK	943
GBK	1386

Table 569. Codeset names and CCSIDs (continued)	
Codeset names	CCSIDs
koi8-r	878

 A complete list of conversions provided is shown in [Table 571 on page 926](#).

National languages

This information contains languages supported by IBM MQ.

The languages supported by IBM MQ are:

- US English - see topic [“US English” on page 903](#)
- German - see topic [“German” on page 903](#)
- Danish and Norwegian - see topic [“Danish and Norwegian” on page 904](#)
- Finnish and Swedish - see topic [“Finnish and Swedish” on page 904](#)
- Italian - see topic [“Italian” on page 905](#)
- Spanish - see topic [“Spanish” on page 906](#)
- UK English / Gaelic - see topic [“UK English /Gaelic” on page 906](#)
- French - see topic [“French” on page 907](#)
- Multilingual - see topic [“Multilingual” on page 908](#)
- Portuguese - see topic [“Portuguese” on page 908](#)
- Icelandic - see topic [“Icelandic” on page 909](#)
- Eastern European languages - see topic [“Eastern European languages” on page 910](#)
- Cyrillic - see topic [“Cyrillic” on page 911](#)
- Estonian - see topic [“Estonian” on page 911](#)
- Latvian and Lithuanian - see topic [“Latvian and Lithuanian” on page 912](#)
- Ukrainian - see topic [“Ukrainian” on page 914](#)
- Greek - see topic [“Greek” on page 914](#)
- Turkish - see topic [“Turkish” on page 915](#)
- Hebrew - see topic [“Hebrew” on page 915](#)
- Farsi - see topic [“Farsi” on page 917](#)
- Urdu - see topic [“Urdu” on page 917](#)
- Thai - see topic [“Thai” on page 918](#)
- Lao - see topic [“Lao” on page 918](#)
- Vietnamese - see topic [“Vietnamese” on page 918](#)
- Japanese Latin SBCS - see topic [“Japanese Latin SBCS” on page 918](#)
- Japanese Katakana SBCS - see topic [“Japanese Katakana SBCS” on page 920](#)
- Japanese Kanji/ Latin Mixed - see topic [“Japanese Kanji/ Latin Mixed” on page 921](#)
- Japanese Kanji/ Katakana Mixed - see topic [“Japanese Kanji/ Katakana Mixed” on page 923](#)
- Korean - see topic [“Korean” on page 924](#)
- Simplified Chinese - see topic [“Simplified Chinese” on page 924](#)
- Traditional Chinese - see topic [“Traditional Chinese” on page 925](#)

US English

Details of CCSIDs and CCSID conversion for US English.

The following table shows the native CCSIDs for US English on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	37, 924, 1140
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 1252, 5348, 858
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

37

Does not convert to code pages 923, 858

924

Does not convert to code pages 437, 858, 1051, 1140, 1252, 1275, 5348

1140

Does not convert to code pages 924, 1051, 1275

German

Details of CCSIDs and CCSID conversion for German.

The following table shows the native CCSIDs for German on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	273, 924, 1141
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

273

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 273, 437, 858, 1051, 1141, 1252, 1275, 5348

1141

Does not convert to code pages 924, 1051, 1275

Danish and Norwegian

Details of CCSIDs and CCSID conversion for Danish and Norwegian.

The following table shows the native CCSIDs for Danish and Norwegian on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	277, 924, 1142
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	850, 858, 865, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

277

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 277, 858, 865, 1051, 1142, 1252, 1275, 5348

1142

Does not convert to code pages 924, 865, 1051, 1275

AIX

Code page:

819

Does not convert to code page 865

HP-UX

Code page:

1051

Does not convert to code page 865

Windows

Code page:

865

Does not convert to code pages 1051, 1275

Finnish and Swedish

Details of CCSIDs and CCSID conversion for Finnish and Swedish.

The following table shows the native CCSIDs for Finnish and Swedish on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	278, 924, 1143
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 865, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

278

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 278, 437, 858, 865, 1051, 1143, 1252, 1275, 5348

1143

Does not convert to code pages 865, 924, 1051, 1275

AIX

Code page:

819

Does not convert to code page 865

850

Does not convert to code page 865

HP-UX

Code page:

1051

Does not convert to code page 865

Windows

Code page:

865

Does not convert to code pages 1051, 1275

Italian

Details of CCSIDs and CCSID conversion for Italian.

The following table shows the native CCSIDs for Italian on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	280, 924, 1144
AIX	819, 923, 5348
HP-UX	819, 923, 1051

Platform	Native CCSIDs
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

280

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 280, 437, 858, 1051, 1144, 1252, 1275, 5348

1144

Does not convert to code pages 924, 1051, 1275

Spanish

Details of CCSIDs and CCSID conversion for Spanish.

The following table shows the native CCSIDs for Spanish on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	284, 924, 1145
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

284

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 284, 437, 858, 1051, 1145, 1252, 1275, 5348

1145

Does not convert to code pages 924, 1051, 1275

UK English /Gaelic

Details of CCSIDs and CCSID conversion for UK English/Gaelic.

The following table shows the native CCSIDs for UK English / Gaelic on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	285, 924, 1146
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

285

Does not convert to code pages 858, 923, 924, 1275

924

Does not convert to code pages 285, 437, 858, 1051, 1146, 1252, 1275, 5348

1146

Does not convert to code pages 924, 1051, 1275

French

Details of CCSIDs and CCSID conversion for French.

The following table shows the native CCSIDs for French on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	297, 924, 1147
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

297

Does not convert to code pages 858, 923, 924, 1275, 5348

924

Does not convert to code pages 297, 437, 858, 1051, 1147, 1252, 1275, 5348

1147

Does not convert to code pages 924, 1051, 1275

Multilingual

Details of CCSIDs and CCSID conversion for Multilingual.

The following table shows the native CCSIDs for multilingual conversion on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	500, 924, 1148
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	437, 850, 858, 1252, 5348
HP Integrity NonStop Server, SINIX, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

500

Does not convert to code pages 858, 923

924

Does not convert to code pages 437, 858, 1051, 1148, 1252, 1275, 5348

1148

Does not convert to code pages 924, 1051, 1275

Portuguese

Details of CCSIDs and CCSID conversion for Portuguese.

The following table shows the native CCSIDs for Portuguese on supported platforms:

Platform	Native CCSIDs
IBM i	37, 500, 924, 1140
z/OS	500, 924, 1140
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	850, 858, 860, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

37

Does not convert to code pages 858, 923, 1275

500

Does not convert to code pages 858, 923, 1275

924

Does not convert to code pages 858, 860, 1051, 1140, 1252, 1275, 5348

1140

Does not convert to code pages 860, 924, 1051, 1275

HP-UX

Code page:

1051

Does not convert to code page 860

Windows

Code page:

860

Does not convert to code pages 1051, 1275

Icelandic

Details of CCSIDs and CCSID conversion for Icelandic.

The following table shows the native CCSIDs for Icelandic on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	871, 924, 1149
AIX	819, 923, 5348
HP-UX	819, 923, 1051
Windows	850, 858, 861, 1252, 5348
HP Integrity NonStop Server, Solaris, Linux	819, 923
Apple client	1275

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

871

Does not convert to code pages 858, 923, 924, 1275, 5348

924

Does not convert to code pages 858, 861, 871, 1051, 1149, 1252, 1275, 5348

1149

Does not convert to code pages 924, 1051, 1275

HP-UX

Code page:

1051

Does not convert to code page 861

Windows

Code page:

861

Does not convert to code pages 1051, 1275

Eastern European languages

Details of CCSIDs and CCSID conversion for Eastern European Languages. The typical languages using these CCSIDs include Albanian, Croatian, Czech, Hungarian, Polish, Romanian, Serbian, Slovak, and Slovenian.

The following table shows the native CCSIDs for Eastern European languages on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	870, 1153
Windows	852, 1250, 5346, 9044
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux	912
Eastern European Apple client	1282
Romanian Apple client	1285
Croatian Apple client	1284

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

z/OS

Code page:

870

Does not convert to code pages 1284, 1285

1153

Does not convert to code pages 1250, 1284, 1285

IBM i

Code page:

870

Does not convert to code pages 1284, 1285, 5346, 9044

1153

Does not convert to code pages 1282, 1284, 1285, 5346, 9044

HP-UX, Solaris, Linux

Code page:

912

Does not convert to code pages 1284, 1285

HP Integrity NonStop Server

Code page:

912

Does not convert to code pages 1153, 1284, 1285, 9044

Windows

Code page:

852

Does not convert to code pages 1284, 1285

1250

Does not convert to code pages 1284, 1285

9044

Does not convert to code pages 912, 1282, 1284, 1285

Cyrillic

Details of CCSIDs and CCSID conversion for Cyrillic. The typical languages using these CCSIDs include Belarussian, Bulgarian, Macedonian, Russian, and Serbian.

The following table shows the native CCSIDs for Cyrillic on supported platforms:

Platform	Native CCSIDs
z/OS	1025
IBM i	880, 1025
Windows	855, 866, 1131, 1251, 5347
Solaris	878, 915
AIX, HP-UX, Linux , HP Integrity NonStop Server	915
Apple client	1283

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

880

Does not convert to code pages 855, 866, 878, 1131, 5347

1025

Does not convert to code pages 878, 5347

Windows

Code page:

855

Does not convert to code page 1131

866

Does not convert to code page 1131

1131

Does not convert to code pages 855, 866, 880, 1283

Estonian

Details of CCSIDs and CCSID conversion for Estonian.

The following table shows the native CCSIDs for Estonian on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1122, 1157

Platform	Native CCSIDs
Windows	902, 922, 1257, 5353, 9449
AIX, HP-UX, Solaris, Linux	902, 922
HP Integrity NonStop Server	922

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

1122

Does not convert to code pages 902, 1157, 9449

1157

Does not convert to code pages 922, 1122, 1257, 9449

IBM i

Code page:

1122

Does not convert to code pages 902, 5353, 9449

1157

Does not convert to code pages 922, 5353, 9449

HP-UX, Solaris, Linux

Code page:

902

Does not convert to code pages 922, 1122, 9449

922

Does not convert to code pages 902, 1157, 9449

Windows

Code page:

5353

Does not convert to code page 9449

9449

Does not convert to code pages 902, 922, 1122, 1157, 1257, 5353

902

Does not convert to code pages 922, 1122, 9449

HP Integrity NonStop Server

Code page:

922

Does not convert to code pages 902, 1157, 9449

Latvian and Lithuanian

Details of CCSIDs and CCSID conversion for Latvian and Lithuanian.

The following table shows the native CCSIDs for Latvian and Lithuanian on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1112, 1156
Windows	901, 921, 1257, 5353, 9449
AIX, HP-UX, Solaris, Linux	901, 921
HP Integrity NonStop Server,	921

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

1112

Does not convert to code pages 901, 1156, 9449

1156

Does not convert to code pages 901, 1156, 9449

IBM i

Code page:

1112

Does not convert to code page 5353

1153

Does not convert to code pages 921, 5353, 9449

HP-UX, Solaris, Linux

Code page:

902

Does not convert to code pages 921, 1112, 1257, 9449

921

Does not convert to code pages 901, 1156, 9449

Windows

Code page:

901

Does not convert to code pages 921, 1112, 1257, 9449

5355

Does not convert to code page 9449

9449

Does not convert to code pages 901, 921, 1112, 1156, 1257

HP Integrity NonStop Server

Code page:

921

Does not convert to code pages 901, 1156, 9449

Ukrainian

Details of CCSIDs and CCSID conversion for Ukrainian.

The following table shows the native CCSIDs for Ukrainian on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1123
Windows	1124, 1125, 1251, 5347
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux	1124

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

IBM i

Code page:

1123

Does not convert to code page 5347

HP-UX

Code page:

1124

Does not convert to code page 5347

Windows

Code page:

1125

Does not convert to code page 1123

Greek

Details of CCSIDs and CCSID conversion for Greek.

The following table shows the native CCSIDs for Greek on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	875
HP-UX	813 (see note)
Windows	869, 1253, 5349
AIX, NCR, HP Integrity NonStop Server, Solaris, Linux	813
Apple client	1280
DOS client	737

Note: Only the ISO codeset is supported on HP-UX. The HP-UX proprietary greek8 codeset has no registered CCSID and is not supported.

All non-client platforms support conversion between their native CCSIDs, the native CCSIDs of the other platforms with the following exceptions.

IBM i

Code page:

875

Does not convert to code page 5349

Windows

Code page:

1253

Does not convert to code page 737

5349

Does not convert to code page 737

Turkish

Details of CCSIDs and CCSID conversion for Turkish.

The following table shows the native CCSIDs for Turkish on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1026
HP-UX	920 (see note)
Windows	857, 1254, 5350
AIX, HP Integrity NonStop Server, Solaris, Linux	920
Apple client	1281

Note: Only the ISO codeset is supported on HP-UX. The HP-UX proprietary turkish8 codeset has no registered CCSID and is not supported.

All non-client platforms support conversion between their native CCSIDs and the native CCSIDs of the other platforms, with the following exceptions.

IBM i

Code page:

1026

Does not convert to code page 5350

Hebrew

Details of CCSIDs and CCSID conversion for Hebrew.

The following table shows the native CCSIDs for Hebrew on supported platforms:

Platform	Native CCSIDs
z/OS	424, 803, 4899, 12712
IBM i	424
AIX	916, 9048
HP-UX	916 (see note)
Windows	1255, 5351
HP Integrity NonStop Server, Solaris, Linux	916

Note: Only the ISO codeset is supported on HP-UX. The HP-UX proprietary greek8 codeset has no registered CCSID and is not supported.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

424

Does not convert to code pages 867, 4899, 9048, 12712

803

Does not convert to code pages 867, 4899, 5351, 9048, 12712

4899

Does not convert to code pages 424, 803, 856, 862, 916, 1255

12712

Does not convert to code pages 424, 803, 856, 916, 1255

IBM i

Code page:

424

Does not convert to code pages 803, 867, 4899, 5351, 9048, 12712

Code page 424 also converts to and from CCSID 4952, which is a variant of 856.

AIX

Code page:

916

Does not convert to code pages 867, 4899, 9048, 12712

9048

Does not convert to code pages 424, 803, 856, 862, 916, 1255

Windows

Code page:

1255

Does not convert to code pages 867, 4899, 9048, 12712

5351

Does not convert to code page 803

Arabic

Details of CCSIDs and CCSID conversion for Arabic

The following table shows the native CCSIDs for Arabic on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	420
AIX	1046, 1089
HP-UX	1089 (see note)
Windows	720, 864, 1256, 5352
HP Integrity NonStop Server, Solaris, Linux	1089

Note: Only the ISO codeset is supported on HP-UX. The HP-UX proprietary arabic8 codeset has no registered CCSID and is not supported.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

IBM i

Code page:

420

Does not convert to code page 5352

HP-UX, Solaris, Linux, HP Integrity NonStop Server, Tru64

Code page:

1089

Does not convert to code page 720

Windows

Code page:

720

Does not convert to code pages 1089, 5352

5352

Does not convert to code page 720

Farsi

Details of CCSIDs and CCSID conversion for Farsi.

The following table shows the native CCSIDs for Farsi on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1097
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux Windows	1098 (see note)

Note: The native CCSID for these platforms has not been standardized and might change.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms.

Urdu

Details of CCSIDs and CCSID conversion for Urdu.

The following table shows the native CCSIDs for Urdu on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	918
Windows	868
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux	1006

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

IBM i

Code page:

918

Does not convert to code page 1006

Thai

Details of CCSIDs and CCSID conversion for Thai.

The following table shows the native CCSIDs for Thai on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	838
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux Windows	874 (see note)

Note: The native CCSID for these platforms has not been standardized and might change.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms.

Lao

Details of CCSIDs and CCSID conversion for Lao.

The following table shows the native CCSIDs for Lao on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1132
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux Windows	1133

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms.

Vietnamese

Details of CCSIDs and CCSID conversion for Vietnamese.

The following table shows the native CCSIDs for Vietnamese on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1130
Windows	1258, 5354
AIX, HP-UX, HP Integrity NonStop Server, Solaris, Linux	1129

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

IBM i

Code page:

1130

Does not convert to code pages 1129, 5354

Japanese Latin SBCS

Details of CCSIDs and CCSID conversion for Japanese Latin SBCS.

The following table shows the native CCSIDs for Japanese Latin SBCS on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1027
AIX	932, 5050, 33722 (see Note 1)
Windows	932, 943 (see Notes 2 and 3)

Platform	Native CCSIDs
Linux, HP Integrity NonStop Server, Solaris	943, 5050
HP-UX	Not known

Note:

1. 5050 and 33722 are CCSIDs related to base code page 954 on AIX. The CCSID reported by the operating system is 33722.
2. Windows NT uses code page 932 but this is best represented by the CCSID of 943. However, not all platforms of IBM MQ support this CCSID.

On IBM MQ for Windows CCSID 932 is used to represent code page 932, but a change to file `./conv/table/ccsid.tbl` can be made which changes the CCSID used to 943.

3. IBM MQ does not support code pages based on the JIS X 0213 (JIS2004) standard.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

1027

Does not convert to code pages 932, 942, 943, 954, 5050, 33722

IBM i

Code page:

1027

Does not convert to code page 932

AIX

Code page:

932

Does not convert to code page 1027

5050

Does not convert to code page 1027

33722

Does not convert to code page 1027

Linux

Code page:

943

Does not convert to code page 1027

5050

Does not convert to code page 1027

Solaris

Code page:

943

Does not convert to code page 1027

5050

Does not convert to code page 1027

HP Integrity NonStop Server

Code page:

943

Does not convert to code page 1027

5050

Does not convert to code page 1027

Japanese Katakana SBCS

Details of CCSIDs and CCSID conversion for Japanese Katakana SBCS.

The following table shows the native CCSIDs for Japanese Katakana SBCS on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	290
HP-UX	897
AIX	932, 5050, 33722 (see Note 1)
Windows	932, 943 (see Notes 2 and 3)
Linux, HP Integrity NonStop Server, Solaris	943, 5050

Note:

1. 5050 and 33722 are CCSIDs related to base code page 954 on AIX. The CCSID reported by the operating system is 33722.
2. Windows NT uses code page 932 but this is best represented by the CCSID of 943. However, not all platforms of IBM MQ support this CCSID.
On IBM MQ for Windows CCSID 932 is used to represent code page 932, but a change to file `.. / conv/table/ccsid.tbl` can be made which changes the CCSID used to 943.
3. IBM MQ does not support code pages based on the JIS X 0213 (JIS2004) standard.
4. In addition to the previous conversions, the IBM MQ products on AIX, HP-UX, Solaris, Linux and Tru64 support conversion from CCSID 897 to CCSIDs 37, 273, 277, 278, 280, 284, 285, 290, 297, 437, 500, 819, 850, 1027, and 1252.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

290

Does not convert to code pages 932, 943, 954, 5050, 33722

IBM i

Code page:

290

Does not convert to code page 932

AIX

Code page:

932

Does not convert to code pages 290, 897

5050

Does not convert to code pages 290, 897

33722

Does not convert to code pages 290, 897

HP-UX

Code page:

897

Does not convert to code pages 932, 943, 954, 5050, 33722

Linux

Code page:

943

Does not convert to code pages 290, 897

5050

Does not convert to code pages 290, 897

Solaris

Code page:

943

Does not convert to code pages 290, 897

5050

Does not convert to code pages 290, 897

HP Integrity NonStop Server

Code page:

943

Does not convert to code pages 290, 897

5050

Does not convert to code pages 290, 897

Japanese Kanji/ Latin Mixed

Details of CCSIDs and CCSID conversion for Japanese Kanji/Latin Mixed.

The following table shows the native CCSIDs for Japanese Kanji/ Latin Mixed on supported platforms:

Platform	Native CCSIDs
IBM i, z/OS	1399, 5035 (see Note 1)
AIX	932, 5050, 33722 (see Note 2)
HP-UX	932, 954, 5039 (see Note 3)
Windows	932, 943 (see Notes 4 and 5)
Linux, HP Integrity NonStop Server, Solaris	943, 5050

Note:

1. 5035 is a CCSID related to code page 939
2. 5050 and 33722 are CCSIDs related to base code page 954 on AIX. The CCSID reported by the operating system is 33722.
3. Code sets japan15 and SJIS on HP-UX are represented by CCSID 932. These have a few DBCS characters having different representations in SJIS so 932 may be converted incorrectly if the conversion is not performed on an HP-UX system. IBM MQ for HP-UX supports 5039, the correct CCSID for HP SJIS. A change to file `/var/mqm/conv/ccsid.tbl` can be made to change the CCSID used from 932 to 5039.
4. Windows NT uses code page 932 but this is best represented by the CCSID of 943. However, not all platforms of IBM MQ support this CCSID.

On IBM MQ for Windows CCSID 932 is used to represent code page 932, but a change to file `./conv/table/ccsid.tbl` can be made which changes the CCSID used to 943.
5. IBM MQ does not support code pages based on the JIS X 0213 (JIS2004) standard.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

1399

Does not convert to code pages 954, 5035, 5050, 33722

5035

Does not convert to code pages 954, 1399, 5050, 33722

IBM i

Code page:

1399

Does not convert to code page 5039

5035

Does not convert to code page 5039

HP-UX

Code page:

932

Does not convert to code pages 942, 943, 1399

954

Does not convert to code pages 942, 943, 1399

5039

Does not convert to code pages 942, 943, 1399

HP Integrity NonStop Server

Code page:

943

Does not convert to code page 1399

5050

Does not convert to code page 1399

Japanese Kanji/ Katakana Mixed

Details of CCSIDs and CCSID conversion for Japanese Kanji/Katakana Mixed.

Table 570. Native CCSIDs for Japanese Kanji/ Katakana Mixed on supported platforms	
Platform	Native CCSIDs
z/OS	1390, 5026 (see Note 1)
IBM i	5026 (see Note 1)
AIX	932, 5050, 33722 (see Note 2)
HP-UX	932, 954, 5039 (see Note 3)
Windows	932, 943 (see Notes 4 and 5)
Linux, HP Integrity NonStop Server, Solaris	943, 5050

Note:

1. The single-byte mode of CCSIDs 1390 and 5026 in EBCDIC contain lower-case characters in different locations to the typical/invariant layout for basic Latin and care must be taken to ensure data is not lost when message data is being converted to other CCSIDs. In addition, the use of these CCSIDs as a queue manager's default CCSID may cause issues when communicating with other queue managers, for example, channel names using lower case characters may not be correctly interpreted on the remote system. 5026 is a CCSID related to code page 930. CCSID 5026 is the CCSID reported on IBM i when the Japanese Katakana (DBCS) feature is selected.
2. 5050 and 33722 are CCSIDs related to base code page 954 on AIX. The CCSID reported by the operating system is 33722.
3. Code sets japan15 and SJIS on HP-UX are represented by CCSID 932. These have a few DBCS characters having different representations in SJIS so 932 may be converted incorrectly if the conversion is not performed on an HP-UX system. IBM MQ for HP-UX supports 5039, the correct CCSID for HP SJIS. A change to file `/var/mqm/conv/ccsid.tbl` can be made to change the CCSID used from 932 to 5039.
4. Windows NT uses code page 932 but this is best represented by the CCSID of 943. However, not all platforms of IBM MQ support this CCSID.

On IBM MQ for Windows, CCSID 932 is used to represent code page 932, but a change to file `./conv/table/ccsid.tbl` can be made that changes the CCSID used to 943.
5. IBM MQ does not support code pages based on the JIS X 0213 (JIS2004) standard.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

1390

Does not convert to code pages 954, 5026, 5050, 33722

Does not accept lowercase characters.

5026

Does not convert to code pages 954, 1390, 5050, 33722

IBM i

Code page:

5026

Does not convert to code pages 1390, 5039

HP-UX

Code page:

932

Does not convert to code pages 942, 943, 1390

954

Does not convert to code pages 942, 943, 1390

5039

Does not convert to code pages 942, 943, 1390

HP Integrity NonStop Server

Code page:

943

Does not convert to code page 1390

5050

Does not convert to code page 1390

Korean

Details of CCSIDs and CCSID conversion for Korean.

The following table shows the native CCSIDs for Korean on supported platforms:

Platform	Native CCSIDs
z/OS, IBM i	933, 1364
AIX, HP-UX, Linux, HP Integrity NonStop Server, Solaris	970
Windows	949, 1363

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

933

Does not convert to code page 970

1364

Does not convert to code page 970

HP-UX

Code page:

970

Does not convert to code pages 949, 1363, 1364

Simplified Chinese

Details of CCSIDs and CCSID conversion for Simplified Chinese.

The following table shows the native CCSIDs for Simplified Chinese on supported platforms:

Platform	Native CCSIDs
z/OS	935, 1388
IBM i	935, 1388

Platform	Native CCSIDs
AIX	1383, 1386
HP-UX	1381 (see Note 1)
Windows	1381, 1386(see Note 2)
Linux, HP Integrity NonStop Server, Solaris	1383

Note:

1. Code sets prc15 and hp15CN on HP-UX are represented by CCSID 1381.
2. Windows uses code page 936 but this is best represented by the CCSID of 1386. However, not all platforms of IBM MQ support this CCSID.

On IBM MQ for Windows CCSID 1381 is used to represent code page 936, but a change to file `./conv/table/ccsid.tbl` can be made which changes the CCSID used to 1386.

3. IBM MQ supports phase one of the Chinese GB18030 standard.

On z/OS, Linux, Windows, and Solaris, conversion support is provided between Unicode (UTF-8 and UCS-2) and CCSID 1388 (EBCDIC with GB18030 extensions), Unicode (UTF-8 and UCS-2) and CCSID 5488 (GB18030 phase one), and between CCSID 1388 and CCSID 5488.

Note:

On IBM i, support is provided by the operating system for conversion between Unicode (UTF-8 and UCS-2) and CCSID 1388 (EBCDIC with GB18030 extensions).

On HP-UX there is currently no support available on the HP11 operating system for GB18030.

On HP11i, patch PHCO_26456 provides conversion support between GB18030 (CCSID 5488) and Unicode. Support is not provided for the conversion between GB18030 and 1388 (EBCDIC).

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

935

Does not convert to code page 1383

1388

Does not convert to code page 1383

HP-UX

Code page:

1381

Does not convert to code pages 1383, 1386, 1388

Traditional Chinese

Details of CCSIDs and CCSID conversion for Traditional Chinese.

The following table shows the native CCSIDs for Traditional Chinese on supported platforms:

Platform	Native CCSIDs
z/OS, IBM i	937
HP-UX	938, 950, 964 (see Note)
Windows	950

Platform	Native CCSIDs
AIX, HP Integrity NonStop Server, Solaris, Linux	950, 964

Note: Code set roc15 on HP-UX is represented by CCSID 938.

All platforms support conversion between their native CCSIDs and the native CCSIDs of other platforms, with the following exceptions.

z/OS

Code page:

937

Does not convert to code page 964

1388

Does not convert to code page 1383

HP-UX

Code page:

938

Does not convert to code page 948

950

Does not convert to code page 948

964

Does not convert to code page 948

Linux, Solaris

Code page:

964

Does not convert to code page 938

z/OS conversion support

A list of supported CCSID conversions.

Table 571. IBM MQ for z/OS CCSID conversion support	
CCSID	Converts to and from CCSIDs
37	256, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 720, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-866, 869-871, 874-875, 880, 897, 903-905, 912, 914-916, 920-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1097, 1100, 1112, 1114-1115, 1122, 1124, 1126, 1130-1132, 1137, 1140-1149, 1200, 1208, 1250-1255, 1257-1258, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5210-5211, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25480, 25617, 25619, 25664, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
256	37, 273, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 737, 775, 819, 833, 836, 838, 850, 852, 857, 860-866, 869-871, 875, 880, 905, 1025-1027, 1112, 1122, 1200, 1208, 1251-1252, 1275, 4386, 4929, 4932, 4934, 4946, 4948, 4953, 4960, 4971, 5123, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 13121, 13488, 16804, 17248, 17584, 28709
259	437, 808, 850-852, 855-858, 860-865, 867, 869, 872, 874, 899, 901-902, 915, 1098, 1161-1162, 1200, 1208, 1250-1258, 4946, 4948, 4951-4953, 4960, 4970, 5346, 5348, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584
273	37, 256, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1250, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951-4953, 4960, 4970-4971, 5012, 5123, 5346, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
274	500, 1047
275	37, 437, 500, 819, 850, 1047, 1200, 1208, 1252, 4946, 5348, 8229, 13488, 17584, 28709
277	37, 256, 273, 278, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
278	37, 256, 273, 277, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
280	37, 256, 273, 277-278, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
281	1047
282	500, 1047, 1200, 1208, 13488, 17584

Table 571. IBM MQ for z/OS CCSID conversion support (continued)	
CCSID	Converts to and from CCSIDS
284	37, 256, 273, 277-278, 280, 285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
285	37, 256, 273, 277-278, 280, 284, 290, 297, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
290	37, 256, 273, 277-278, 280, 284-285, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 895-897, 1009, 1025-1027, 1040-1043, 1047, 1088, 1112, 1122, 1139, 1200, 1208, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 4992, 5123, 8229, 8482, 9025, 9044, 9049, 9056, 13121, 13488, 17248, 17584, 25473, 25617, 25619, 25664, 28709
293	1200, 1208, 13488, 17584
297	37, 256, 273, 277-278, 280, 284-285, 290, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
300	301, 941, 1200, 1208, 1351, 4396, 8492, 13488, 16684, 17584
301	300, 941, 1200, 1208, 1351, 4396, 8492, 13488, 16684, 17584
367	37, 256, 273, 277-278, 280, 284, 290, 297, 500, 819, 833, 836, 850, 871, 875, 1009, 1026-1027, 1041, 1088, 1115, 1126, 1200, 1208, 4386, 4929, 4932, 4946, 4971, 5123, 5211, 8229, 8482, 9025, 13121, 13488, 17584, 25617, 25664, 28709
420	37, 256, 424, 437, 500, 720, 737, 775, 819, 850, 852, 857, 860-865, 1008, 1046, 1089, 1098, 1112, 1122, 1127, 1200, 1208, 1252, 1256, 4946, 4948, 4953, 4960, 5104, 5142, 5352, 8229, 8612, 9044, 9049, 9056, 9238, 13488, 16804, 17248, 17584, 28709
423	37, 256, 273, 277-278, 280, 284-285, 297, 437, 500, 737, 775, 813, 819, 838, 850-852, 857, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1041-1043, 1112, 1122, 1200, 1208, 1252-1253, 1280, 4909, 4934, 4946, 4948, 4953, 4960, 4970-4971, 5012, 5123, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
424	37, 256, 420, 437, 500, 737, 775, 803, 819, 836, 850, 852, 856-857, 860-865, 916, 1112, 1122, 1200, 1208, 1252, 1255, 4932, 4946, 4948, 4952-4953, 4960, 5012, 5351, 8229, 8612, 9044, 9049, 9056, 13488, 16804, 17248, 17584, 28709
437	37, 256, 259, 273, 275, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-863, 865-866, 869-871, 874-875, 880, 897, 903, 905, 912, 914-916, 920-924, 1025-1027, 1040-1043, 1047, 1051, 1097, 1098, 1114-1115, 1126, 1140-1149, 1200, 1208, 1252, 1257, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5210-5211, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 28709
500	37, 256, 273-275, 277-278, 280, 282, 284-285, 290, 297, 367, 420, 423-424, 437, 737, 775, 813, 819, 833, 836, 838, 850-852, 855-858, 860-866, 869-871, 874-875, 880, 891, 895, 897, 903-905, 912, 914-916, 920-924, 1004, 1009-1021, 1023, 1025-1027, 1040-1043, 1046-1047, 1051, 1088-1089, 1097, 1100-1107, 1112, 1114-1115, 1122, 1124-1126, 1129-1133, 1137, 1140-1149, 1200, 1208, 1250-1258, 1275, 1280-1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951-4953, 4960, 4970-4971, 5012, 5123, 5142, 5210-5211, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 9238, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25480, 25617, 25619, 25664, 28709
720	37, 420, 864, 1200, 1208, 1256, 4960, 8229, 8612, 9056, 13488, 16804, 17248, 17584, 28709
737	37, 256, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 833, 836, 838, 850, 869-871, 875, 880, 905, 1025-1027, 1097, 1200, 1208, 1252-1253, 1280, 4386, 4909, 4929, 4932, 4934, 4946, 4971, 5123, 8229, 8482, 8612, 9025, 9030, 9061, 13121, 13488, 16804, 17584, 28709
775	37, 256, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 833, 836, 838, 850, 870-871, 875, 880, 905, 1025-1027, 1097, 1112, 1122, 1200, 1208, 1252, 1257, 4386, 4929, 4932, 4934, 4946, 4971, 5123, 8229, 8482, 8612, 9025, 9030, 13121, 13488, 16804, 17584, 28709
803	424, 819, 850, 856, 862, 916, 1200, 1208, 1252, 1255, 4946, 4952, 5012, 13488, 17584
806	1200, 1208, 13488, 17584
808	259, 858-859, 872, 923-924, 1140, 1148, 1153-1154, 1200, 1208, 5347, 5348, 13488, 17584
813	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252-1253, 1280, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
819	37, 256, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 803, 813, 833, 836, 838, 850, 852, 855, 857-858, 860-861, 863-866, 869-871, 874-875, 880, 897, 903, 905, 912, 914-916, 920-924, 1004, 1025-1027, 1041-1043, 1047, 1051, 1088-1089, 1097, 1098, 1112, 1114, 1122-1123, 1126, 1130, 1132, 1137, 1140-1149, 1200, 1208, 1250-1255, 1257-1258, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5210, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
833	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 836, 850, 852, 855, 857, 860-865, 870-871, 891, 1009, 1025-1027, 1040-1043, 1088, 1112, 1122, 1126, 1200, 1208, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 5123, 8229, 8482, 9025, 9044, 9049, 9056, 13121, 13488, 17248, 17584, 25617, 25619, 25664, 28709
834	926, 951, 1200, 1208, 1362, 4930, 9026, 13488, 17584
835	927, 947, 1200, 1208, 4931, 9027, 13488, 17584, 21427
836	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 424, 437, 500, 737, 775, 819, 833, 850, 852, 855, 857, 870-871, 875, 903, 1009, 1025-1027, 1040-1043, 1088, 1112, 1114-1115, 1122, 1200, 1208, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4971, 5123, 5210-5211, 8229, 8482, 9025, 9044, 9049, 13121, 13488, 17584, 25479, 25617, 25619, 25664, 28709
837	928, 1200, 1208, 1380, 1385, 4933, 13488, 17584
838	37, 256, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 775, 813, 819, 850, 852, 857, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1112, 1122, 1200, 1208, 1252, 4909, 4934, 4946, 4948, 4953, 4960, 4970-4971, 5012, 5123, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 28709
848	924, 1148, 1158, 1200, 1208, 5347, 13488, 17584
849	924, 1148, 1154, 1200, 1208, 5347, 13488, 17584
850	37, 256, 259, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 737, 775, 803, 813, 819, 833, 836, 838, 852, 855-858, 860-866, 869-871, 874-875, 880, 897, 903, 905, 912, 914-916, 920-924, 1004, 1025-1027, 1040-1043, 1047, 1051, 1088-1089, 1097, 1098, 1100, 1112, 1114, 1122, 1126, 1130, 1132, 1140-1149, 1200, 1208, 1250-1257, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951-4953, 4960, 4970-4971, 5012, 5123, 5210, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
851	259, 423, 500, 875, 1200, 1208, 4971, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
852	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1047, 1088, 1097, 1200, 1208, 1250, 1252, 1282, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5346, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 25664, 28709
855	37, 259, 273, 277-278, 280, 284-285, 290, 297, 437, 500, 819, 833, 836, 850, 852, 857, 866, 870-871, 878, 880, 912, 915, 1025-1027, 1040-1043, 1088, 1200, 1208, 1250-1252, 1283, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 5123, 5346, 5347, 8229, 8482, 9025, 9044, 9049, 13121, 13488, 17584, 25617, 25619, 25664, 28709
856	259, 273, 424, 500, 803, 850, 862, 916, 1200, 1208, 1255, 4946, 4952, 5012, 5351, 13488, 17584
857	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1088, 1097, 1200, 1208, 1252, 1254, 1281, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5350, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 25664, 28709
858	37, 259, 273, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 850, 860-861, 865, 871-872, 901-902, 923-924, 1047, 1051, 1140-1149, 1153-1157, 1160-1162, 1164, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
859	808, 872, 901-902, 1153-1157, 1160-1162, 1164, 1200, 1208, 13488, 17584
860	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 838, 850, 852, 857-858, 861, 863, 865, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 923-924, 1025-1027, 1041-1043, 1097, 1140, 1145-1146, 1148, 1200, 1208, 1252, 4386, 4909, 4929, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 28709
861	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 838, 850, 852, 857-858, 860, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 923-924, 1025-1027, 1041-1043, 1097, 1148, 1149, 1200, 1208, 1252, 4386, 4909, 4929, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 28709
862	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 803, 833, 838, 850, 856, 870-871, 875, 880, 905, 916, 1025-1027, 1097, 1200, 1208, 1252, 1255, 4386, 4929, 4934, 4946, 4952, 4971, 5012, 5123, 5351, 8229, 8482, 8612, 9025, 9030, 12712, 13121, 13488, 16804, 17584, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
863	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 838, 850, 852, 857, 860-861, 865, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1041-1043, 1051, 1097, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 28709
864	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 720, 819, 833, 838, 850, 870-871, 875, 880, 905, 918, 1008, 1025-1027, 1046, 1089, 1097, 1127, 1200, 1208, 1252, 1256, 4386, 4929, 4934, 4946, 4960, 4971, 5104, 5123, 5142, 5352, 8229, 8482, 8612, 9025, 9030, 9056, 9238, 13121, 13488, 16804, 17248, 17584, 28709
865	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 819, 833, 838, 850, 858, 860, 863, 870-871, 875, 880, 905, 923-924, 1025-1027, 1097, 1142-1143, 1148, 1200, 1208, 1252, 4386, 4929, 4934, 4946, 4971, 5123, 5348, 8229, 8482, 8612, 9025, 9030, 13121, 13488, 16804, 17584, 28709
866	37, 256, 437, 500, 819, 850, 855, 870, 878, 880, 915, 1025, 1200, 1208, 1251-1252, 1283, 4946, 4951, 5347, 8229, 13488, 17584, 28709
867	259, 1153-1155, 1160, 1200, 1208, 4899, 5351, 9048, 12712, 13488, 17584
868	918, 1006, 1200, 1208, 13488, 17584
869	37, 256, 259, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 813, 819, 838, 850, 852, 857, 860-861, 863, 870-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252-1254, 1280, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
870	37, 256, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-866, 869, 871, 874-875, 880, 897, 903, 912, 915-916, 920, 1009, 1025-1027, 1040-1043, 1047, 1088, 1112, 1122, 1200, 1208, 1250, 1252, 1282, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5346, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
871	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-865, 869, 870, 874-875, 880, 897, 903, 912, 916, 920, 923-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1112, 1122, 1140-1149, 1200, 1208, 1252, 1275, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5348, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
872	259, 808, 858-859, 923-924, 1140-1149, 1153-1155, 1200, 1208, 5347, 5348, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
874	37, 259, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
875	37, 256, 273, 277-278, 280, 284-285, 297, 367, 423, 437, 500, 737, 775, 813, 819, 836, 838, 850-852, 857, 860-865, 869-871, 874, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1041-1043, 1047, 1088, 1112, 1122, 1200, 1208, 1252-1253, 1280, 4909, 4932, 4934, 4946, 4948, 4953, 4960, 4970-4971, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
878	855, 866, 880, 915, 1025, 1131, 1200, 1208, 1251, 1283, 4951, 5347, 13488, 17584
880	37, 256, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 775, 813, 819, 838, 850, 852, 855, 857, 860-866, 869-871, 874-875, 878, 897, 903, 912, 915-916, 920, 1009, 1025-1027, 1041-1043, 1112, 1122, 1200, 1208, 1251-1252, 1283, 4909, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5347, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 28709
891	500, 833, 1088, 1200, 1208, 4929, 9025, 13121, 13488, 17584, 25664
895	290, 500, 1027, 1041, 1200, 1208, 4386, 5123, 8482, 13488, 17584, 25617
896	290, 1027, 1041, 1200, 1208, 4386, 4992, 5123, 8482, 13488, 17584, 25617
897	37, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252, 4386, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 8229, 8482, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
899	259
901	259, 858-859, 902, 923-924, 1140, 1148, 1156-1157, 1200, 1208, 5348, 5353, 13488, 17584
902	259, 858-859, 901, 923-924, 1140, 1148, 1156-1157, 1200, 1208, 5348, 5353, 13488, 17584
903	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 836, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 912, 916, 920, 1025-1027, 1041-1043, 1115, 1200, 1208, 1252, 4909, 4932, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5211, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
904	37, 500, 1114, 1200, 1208, 5210, 8229, 13488, 17584, 25480, 28709
905	37, 256, 437, 500, 737, 775, 819, 850, 852, 857, 860-865, 920, 1026, 1112, 1122, 1200, 1208, 1252, 1254, 1281, 4946, 4948, 4953, 4960, 8229, 9044, 9049, 9056, 13488, 17248, 17584, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
912	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 916, 920, 1025-1027, 1041-1043, 1047, 1200, 1208, 1250, 1252, 1282, 4909, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5346, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
914	37, 437, 500, 819, 850, 1200, 1208, 1252, 1257, 4946, 8229, 13488, 17584, 28709
915	37, 259, 437, 500, 819, 850, 855, 866, 870, 878, 880, 1025, 1131, 1200, 1208, 1251-1252, 1283, 4946, 4951, 5347, 8229, 13488, 17584, 28709
916	37, 273, 277-278, 280, 284-285, 297, 423-424, 437, 500, 803, 813, 819, 838, 850, 852, 856-857, 860-863, 869-871, 874-875, 880, 897, 903, 912, 920, 1025-1027, 1041-1043, 1200, 1208, 1252, 1255, 4909, 4934, 4946, 4948, 4952-4953, 4970-4971, 5012, 5123, 5351, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
918	864, 868, 1006, 1200, 1208, 4960, 9056, 13488, 17248, 17584
920	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 1025-1026, 1200, 1208, 1252, 1254, 1281, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5350, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 28709
921	37, 437, 500, 819, 850, 922, 1112, 1122, 1200, 1208, 1252, 1257, 4946, 5353, 8229, 13488, 17584, 28709
922	37, 437, 500, 819, 850, 921, 1112, 1122, 1200, 1208, 1252, 1257, 4946, 5353, 8229, 13488, 17584, 28709
923	37, 273, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 850, 858, 860-861, 865, 871-872, 901-902, 924, 1047, 1051, 1140-1149, 1153-1158, 1160-1162, 1164, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
924	37, 273, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 848-850, 858, 860-861, 865, 871-872, 901-902, 923, 1047, 1051, 1140-1149, 1153-1157, 1160-1164, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
926	834, 951, 9026
927	835, 947, 1200, 1208, 4931, 9027, 13488, 17584, 21427
928	837, 1200, 1208, 1380, 13488, 17584
930	931-932, 939, 942-943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
931	930, 932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
932	930-931, 939, 942-943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
933	934, 944, 949, 1200, 1208, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13488, 13651, 17317, 17584, 25510, 25520, 25525, 29616, 29621, 33717, 37813
934	933, 949, 5029, 5045, 5460, 9125, 13221, 17317, 25510, 25525, 29621, 33717, 37813
935	936, 946, 1200, 1208, 1381, 1386, 1388, 5031, 5477, 5482, 5484, 9127, 13223, 13488, 17584, 25512
936	935, 946, 1381, 5031, 5477, 5484, 9127, 13223, 25512
937	938, 948, 950, 1200, 1208, 1370, 5033, 5046, 9142, 13488, 17584, 25514, 25524, 29620
938	937, 950, 1370, 5033, 5046, 9142, 25514
939	930-932, 942-943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
941	300-301, 1200, 1208, 1351, 4396, 8492, 13488, 16684, 17584
942	930-932, 939, 943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
943	930-932, 939, 942, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
944	933, 949, 1200, 1208, 5029, 5045, 5460, 9125, 13221, 13488, 17317, 17584, 25520, 25525, 29616, 29621, 33717, 37813
946	935-936, 1200, 1208, 5031, 5484, 9127, 13223, 13488, 17584, 25512
947	835, 927, 1200, 1208, 4931, 9027, 13488, 17584, 21427
948	937, 950, 1200, 1208, 1370, 5033, 5046, 9142, 13488, 17584, 25524, 29620
949	933-934, 944, 1200, 1208, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13488, 13651, 17317, 17584, 25510, 25520, 25525, 29616, 29621, 33717, 37813
950	937-938, 948, 1200, 1208, 1370, 5033, 5046, 9142, 13488, 17584, 25514, 25524, 29620
951	834, 926, 1200, 1208, 1362, 4930, 9026, 13488, 17584
1004	500, 819, 850, 1200, 1208, 4946, 13488, 17584
1006	868, 918, 1200, 1208, 13488, 17584
1008	420, 864, 1200, 1208, 4960, 5104, 8612, 9056, 13488, 16804, 17248, 17584
1009	37, 273, 277-278, 280, 284, 290, 297, 367, 423, 500, 833, 836, 870-871, 875, 880, 1025-1026, 1200, 1208, 4386, 4929, 4932, 4971, 8229, 8482, 9025, 13121, 13488, 17584, 28709
1010	500, 1200, 1208, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1011	500, 1200, 1208, 13488, 17584
1012	500, 1200, 1208, 13488, 17584
1013	500, 1140, 1200, 1208, 13488, 17584
1014	500, 1200, 1208, 13488, 17584
1015	500, 1200, 1208, 13488, 17584
1016	500, 1200, 1208, 13488, 17584
1017	500, 1200, 1208, 13488, 17584
1018	500, 1200, 1208, 13488, 17584
1019	500, 1200, 1208, 13488, 17584
1020	500
1021	500
1023	500
1025	37, 256, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-866, 869-871, 874-875, 878, 880, 897, 903, 912, 915-916, 920, 1009, 1026-1027, 1040-1043, 1051, 1088, 1112, 1122, 1131, 1200, 1208, 1251-1252, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5347, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
1026	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1009, 1025, 1027, 1040-1043, 1047, 1088, 1112, 1122, 1200, 1208, 1252, 1254, 1281, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5350, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
1027	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874-875, 880, 895-897, 903, 912, 916, 1025-1026, 1040-1043, 1047, 1088, 1112, 1122, 1139, 1200, 1208, 1252, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 4992, 5012, 5123, 8229, 8482, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
1040	37, 273, 277-278, 280, 284-285, 290, 297, 437, 500, 833, 836, 850, 852, 855, 857, 870-871, 1025-1027, 1041-1043, 1088, 1200, 1208, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 5123, 8229, 8482, 9025, 9044, 9049, 13121, 13488, 17584, 25617, 25619, 25664, 28709
1041	37, 273, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 895-897, 903, 912, 916, 1025-1027, 1040, 1042-1043, 1088, 1200, 1208, 1252, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 4992, 5012, 5123, 8229, 8482, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 17584, 25473, 25479, 25617, 25619, 25664, 28709

Table 571. IBM MQ for z/OS CCSID conversion support (continued)

CCSID	Converts to and from CCSIDS
1042	37, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 1025-1027, 1040, 1041, 1043, 1088, 1200, 1208, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 8229, 8482, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 17584, 25473, 25479, 25617, 25619, 25664, 28709
1043	37, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 1025-1027, 1040, 1041, 1042, 1088, 1114, 1200, 1208, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5210, 8229, 8482, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 17584, 25473, 25479, 25617, 25619, 25664, 28709
1046	420, 500, 864, 1089, 1127, 1200, 1208, 1256, 4960, 5142, 5352, 8612, 9056, 9238, 13488, 16804, 17248, 17584
1047	37, 273-275, 277-278, 280, 281, 282, 284-285, 290, 297, 437, 500, 819, 850, 852, 858, 870-871, 875, 912, 923-924, 1026-1027, 1140-1149, 1200, 1208, 1252, 1254, 4946, 4948, 5123, 8229, 8482, 13488, 17584, 28709
1051	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 871, 923-924, 1025, 1097, 1140-1149, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1088	37, 273, 277-278, 280, 284-285, 290, 297, 367, 500, 819, 833, 836, 850, 852, 855, 857, 870-871, 875, 891, 1025-1027, 1040-1043, 1126, 1200, 1208, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4971, 5123, 8229, 8482, 9025, 9044, 9049, 13121, 13488, 17584, 25617, 25619, 25664, 28709
1089	420, 500, 819, 850, 864, 1046, 1127, 1200, 1208, 1256, 4946, 4960, 5142, 5352, 8612, 9056, 9238, 13488, 16804, 17248, 17584
1097	37, 437, 500, 737, 775, 819, 850, 852, 857, 860-865, 1051, 1098, 1112, 1122, 1200, 1208, 1252, 4946, 4948, 4953, 4960, 8229, 9044, 9049, 9056, 13488, 17248, 17584, 28709
1098	259, 420, 437, 819, 850, 1097, 1200, 1208, 1252, 4946, 8612, 13488, 16804, 17584
1100	37, 273, 277-278, 280, 284-285, 297, 500, 850, 4946, 8229, 28709
1101	500
1102	500
1103	500
1104	500
1105	500
1106	500
1107	500

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1112	37, 256, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 775, 819, 833, 836, 838, 850, 870-871, 875, 880, 905, 921-922, 1025-1027, 1097, 1122, 1200, 1208, 1252, 1257, 4386, 4929, 4932, 4934, 4946, 4971, 5123, 5353, 8229, 8482, 8612, 9025, 9030, 13121, 13488, 16804, 17584, 28709
1114	37, 437, 500, 819, 836, 850, 904, 1043, 1115, 1200, 1208, 4932, 4946, 5210-5211, 8229, 13488, 17584, 25480, 25619, 28709
1115	37, 367, 437, 500, 836, 903, 1114, 1200, 1208, 4932, 5210-5211, 8229, 13488, 17584, 25479, 28709
1122	37, 256, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 775, 819, 833, 836, 838, 850, 870-871, 875, 880, 905, 921-922, 1025-1027, 1097, 1112, 1200, 1208, 1252, 1257, 4386, 4929, 4932, 4934, 4946, 4971, 5123, 5353, 8229, 8482, 8612, 9025, 9030, 13121, 13488, 16804, 17584, 28709
1123	819, 1124-1125, 1148, 1200, 1208, 1251-1252, 1283, 5347, 13488, 17584
1124	37, 500, 1123, 1125, 1200, 1208, 1251, 1283, 5347, 8229, 13488, 17584, 28709
1125	500, 1123, 1124, 1200, 1208, 1251, 1283, 5347, 13488, 17584
1126	37, 367, 437, 500, 819, 833, 850, 1088, 1200, 1208, 1252, 4929, 4946, 8229, 9025, 13121, 13488, 17584, 25664, 28709
1127	420, 864, 1046, 1089, 1256, 4960, 5142, 8612, 9056, 9238, 16804, 17248
1129	500, 1130, 1200, 1208, 1258, 5354, 13488, 17584
1130	37, 500, 819, 850, 1129, 1200, 1208, 1252, 1258, 4946, 5354, 8229, 13488, 17584, 28709
1131	37, 500, 878, 915, 1025, 1200, 1208, 1251, 1283, 5347, 8229, 13488, 17584, 28709
1132	37, 500, 819, 850, 1133, 1200, 1208, 1252, 4946, 8229, 13488, 17584, 28709
1133	500, 1132, 1200, 1208, 13488, 17584
1137	37, 500, 819, 1200, 1208, 8229, 13488, 17584, 28709
1139	290, 1027, 4386, 5123, 8482
1140	37, 273, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 850, 858, 860, 863, 871-872, 901-902, 923-924, 1013, 1047, 1051, 1141-1149, 1153-1157, 1160-1162, 1164, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1141	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 871-872, 923-924, 1047, 1051, 1140, 1142-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1142	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 865, 871-872, 923-924, 1047, 1051, 1140-1141, 1143-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1143	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 865, 871-872, 923-924, 1047, 1051, 1140-1142, 1144-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1144	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 871-872, 923-924, 1047, 1051, 1140-1143, 1145-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1145	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 860, 863, 871-872, 923-924, 1047, 1051, 1140-1144, 1146-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1146	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 860, 863, 871-872, 923-924, 1047, 1051, 1140-1145, 1147-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1147	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 871-872, 923-924, 1047, 1051, 1140-1146, 1148-1149, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1148	37, 273, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 848-850, 858, 860-861, 863, 865, 871-872, 901-902, 923-924, 1047, 1051, 1123, 1140-1147, 1149, 1153-1164, 1200, 1208, 1252, 1275, 4899, 4946, 5348, 5349, 8229, 12712, 13488, 17584, 28709
1149	37, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 861, 863, 871-872, 923-924, 1047, 1051, 1140-1148, 1153-1157, 1160-1162, 1200, 1208, 1252, 1275, 4946, 5348, 8229, 13488, 17584, 28709
1153	808, 858-859, 867, 872, 923-924, 1140-1149, 1154-1157, 1160-1162, 1200, 1208, 5348, 9044, 13488, 17584
1154	808, 849, 858-859, 867, 872, 923-924, 1140-1149, 1153, 1155-1157, 1160-1162, 1200, 1208, 5347, 5348, 13488, 17584
1155	858-859, 867, 872, 923-924, 1140-1149, 1153-1154, 1156-1157, 1160-1162, 1200, 1208, 5348, 5350, 9049, 13488, 17584
1156	858-859, 901-902, 923-924, 1140-1149, 1153-1155, 1157, 1160, 1200, 1208, 5348, 5353, 12712, 13488, 17584
1157	858-859, 901-902, 923-924, 1140-1149, 1153-1156, 1160, 1200, 1208, 5348, 5353, 12712, 13488, 17584
1158	848, 923, 1148, 1200, 1208, 5347, 5348, 13488, 17584
1159	1148, 1200, 1208, 13488, 17584
1160	858-859, 867, 923-924, 1140-1149, 1153-1157, 1161-1162, 1200, 1208, 5348, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1161	259, 858-859, 923-924, 1140-1149, 1153-1155, 1160, 5348, 17584
1162	259, 858-859, 923-924, 1140-1149, 1153-1155, 1160, 5348, 17584
1163	924, 1148, 1164, 5354, 17584
1164	858-859, 923-924, 1140, 1148, 1163, 1200, 1208, 5348, 5354, 13488, 17584
1166	1200,1208,13488,17584
1200	37, 256, 259, 273, 275, 277-278, 280, 282, 284-285, 290, 293, 297, 300-301, 367, 420, 423-424, 437, 500, 720, 737, 775, 803, 806, 808, 813, 819, 833-838, 848-852, 855-872, 874-875, 878, 880, 891, 895-897, 901-905, 912, 914-916, 918, 920-924, 927-928, 930, 932-933, 935, 937, 939, 941-944, 946-951, 1004, 1006, 1008-1019, 1025-1027, 1040-1043, 1046-1047, 1051, 1088-1089, 1097-1098, 1112, 1114-1115, 1122-1126, 1129-1133, 1137, 1140-1149, 1153-1160, 1164, 1166, 1208, 1250-1258, 1275-1277, 1280-1285, 1351, 1362-1364, 1370-1371, 1380-1381, 1385-1386, 1388, 1390, 1399, 4899, 4909, 4930, 4933, 4948, 4951-4952, 4960, 4971, 5012, 5039, 5104, 5123, 5142, 5210, 5346-5354, 8482, 8612, 9027, 9030, 9044, 9048-9049, 9056, 9061, 9066, 9238, 12712, 13121, 13218, 13488, 16684, 16804, 17248, 17584, 21427, 28709
1208	37, 256, 259, 273, 275, 277-278, 280, 282, 284-285, 290, 293, 297, 300-301, 367, 420, 423-424, 437, 500, 720, 737, 775, 803, 806, 808, 813, 819, 833-838, 848-852, 855-872, 874-875, 878, 880, 891, 895-897, 901-905, 912, 914-916, 918, 920-924, 927-928, 930, 932-933, 935, 937, 939, 941-944, 946-951, 1004, 1006, 1008-1019, 1025-1027, 1040-1043, 1046-1047, 1051, 1088-1089, 1097-1098, 1112, 1114-1115, 1122-1126, 1129-1133, 1137, 1140-1149, 1153-1160, 1164, 1166, 1200, 1250-1258, 1275-1277, 1280-1285, 1351, 1362-1364, 1370-1371, 1380-1381, 1385-1386, 1388, 1390, 1399, 4899, 4909, 4930, 4933, 4948, 4951-4952, 4960, 4971, 5012, 5026, 5035, 5039, 5104, 5123, 5142, 5210, 5346-5354, 8482, 8612, 9027, 9030, 9044, 9048-9049, 9056, 9061, 9066, 9238, 12712, 13121, 13218, 13488, 16684, 16804, 17248, 17584, 21427, 28709
1250	37, 259, 273, 500, 819, 850, 852, 855, 870, 912, 1200, 1208, 1252, 1282, 4946, 4948, 4951, 5346, 8229, 9044, 13488, 17584, 28709
1251	37, 256, 259, 500, 819, 850, 855, 866, 878, 880, 915, 1025, 1123-1125, 1131, 1200, 1208, 1252, 1283, 4946, 4951, 5347, 8229, 13488, 17584, 28709
1252	37, 256, 259, 273, 275, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 737, 775, 803, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-866, 869-871, 874-875, 880, 897, 903, 905, 912, 914-916, 920-924, 1025-1027, 1041, 1047, 1051, 1097-1098, 1112, 1122-1123, 1126, 1130, 1132, 1140-1149, 1200, 1208, 1250-1251, 1254-1255, 1257, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25617, 28709
1253	37, 259, 423, 500, 737, 813, 819, 850, 869, 875, 1200, 1208, 1280, 4909, 4946, 4971, 5349, 8229, 9061, 13488, 17584, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1254	37, 259, 500, 819, 850, 857, 869, 905, 920, 1026, 1047, 1200, 1208, 1252, 1281, 4946, 4953, 5350, 8229, 9049, 9061, 13488, 17584, 28709
1255	37, 259, 424, 500, 803, 819, 850, 856, 862, 916, 1200, 1208, 1252, 1281, 4946, 4952, 5012, 5351, 8229, 13488, 17584, 28709
1256	259, 420, 500, 720, 850, 864, 1046, 1089, 1127, 1200, 1208, 4946, 4960, 5142, 5352, 8612, 9056, 9238, 13488, 16804, 17248, 17584
1257	37, 259, 437, 500, 775, 819, 850, 914, 921-922, 1112, 1122, 1200, 1208, 1252, 4946, 5353, 8229, 13488, 17584, 28709
1258	37, 259, 500, 819, 1129-1130, 1200, 1208, 5354, 8229, 13488, 17584, 28709
1275	37, 256, 273, 277-278, 280, 284-285, 297, 437, 500, 819, 850, 858, 863, 871, 923-924, 1051, 1140-1149, 1200, 1208, 1252, 4946, 5348, 8229, 13488, 17584, 28709
1276	1200, 1208, 13488, 17584
1277	1200, 1208, 13488, 17584
1280	37, 423, 437, 500, 737, 813, 819, 850, 869, 875, 1200, 1208, 1252-1253, 4909, 4946, 4971, 5349, 8229, 9061, 13488, 17584, 28709
1281	37, 437, 500, 819, 850, 857, 905, 920, 1026, 1200, 1208, 1252, 1254-1255, 4946, 4953, 5350, 8229, 9049, 13488, 17584, 28709
1282	500, 852, 870, 912, 1200, 1208, 1250, 4948, 5346, 9044, 13488, 17584
1283	37, 437, 500, 819, 850, 855, 866, 878, 880, 915, 1025, 1123-1125, 1131, 1200, 1208, 1251-1252, 4946, 4951, 5347, 8229, 13488, 17584, 28709
1284	1200, 1208, 13488, 17584
1285	1200, 1208, 13488, 17584
1351	300-301, 941, 1200, 1208, 4396, 8492, 13488, 16684, 17584
1362	834, 951, 1200, 1208, 4930, 9026, 13488, 17584
1363	933, 949, 1200, 1208, 1364, 5029, 5045, 5460, 9125, 9555, 13221, 13488, 13651, 17317, 17584, 25525, 29621, 33717, 37813
1364	933, 949, 1200, 1208, 1363, 5029, 5045, 5460, 9125, 9555, 13221, 13488, 13651, 17317, 17584, 25525, 29621, 33717, 37813
1370	937-938, 948, 950, 1200, 1208, 1371, 5033, 5046, 9142, 13488, 17584, 25514, 25524, 29620
1371	1200, 1208, 1370, 13488, 17584
1380	837, 928, 1200, 1208, 1385, 4933, 13488, 17584
1381	935-936, 1200, 1208, 1386, 1388, 5031, 5477, 5482, 5484, 9127, 13223, 13488, 17584, 25512
1385	837, 1200, 1208, 1380, 4933, 13488, 17584
1386	935, 1200, 1208, 1381, 1388, 5031, 5477, 5482, 5484, 9127, 13223, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
1388	935, 1200, 1208, 1381, 1386, 5031, 5477, 5482, 5484, 5488, 9127, 13223, 13488, 17584
1390	930-932, 939, 942-943, 1200, 1208, 1399, 5026, 5028, 5035, 5038-5039, 5055, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
1399	930-932, 939, 942-943, 1200, 1208, 1390, 5026, 5028, 5035, 5038-5039, 5050, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
4386	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 895-897, 1009, 1025-1027, 1040-1043, 1088, 1112, 1122, 1139, 1252, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 4992, 5123, 8229, 8482, 9025, 9044, 9049, 9056, 13121, 17248, 25473, 25617, 25619, 25664, 28709
4396	300-301, 941, 1351, 8492, 16684
4899	867, 1148, 1200, 1208, 5351, 9048, 12712, 13488, 17584
4909	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252-1253, 1280, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
4929	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 891, 1009, 1025-1027, 1040-1043, 1088, 1112, 1122, 1126, 1252, 4386, 4932, 4946, 4948, 4951, 4953, 4960, 5123, 8229, 8482, 9025, 9044, 9049, 9056, 13121, 17248, 25617, 25619, 25664, 28709
4930	834, 951, 1200, 1208, 1362, 9026, 13488, 17584
4931	835, 927, 947, 9027, 21427
4932	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 424, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 870-871, 875, 903, 1009, 1025-1027, 1040-1043, 1088, 1112, 1114-1115, 1122, 1252, 4386, 4929, 4946, 4948, 4951, 4953, 4971, 5123, 5210-5211, 8229, 8482, 9025, 9044, 9049, 13121, 25479, 25617, 25619, 25664, 28709
4933	837, 1200, 1208, 1380, 1385, 13488, 17584
4934	37, 256, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 775, 813, 819, 838, 850, 852, 857, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1112, 1122, 1252, 4909, 4946, 4948, 4953, 4960, 4970-4971, 5012, 5123, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 17248, 25473, 25479, 25617, 25619, 28709

Table 571. IBM MQ for z/OS CCSID conversion support (continued)

CCSID	Converts to and from CCSIDS
4946	37, 256, 259, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 737, 775, 803, 813, 819, 833, 836, 838, 850, 852, 855-858, 860-866, 869-871, 874-875, 880, 897, 903, 905, 912, 914-916, 920-924, 1004, 1025-1027, 1040-1043, 1047, 1051, 1088-1089, 1097-1098, 1100, 1112, 1114, 1122, 1126, 1130, 1132, 1140-1149, 1250-1257, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4948, 4951-4953, 4960, 4970-4971, 5012, 5123, 5210, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 16804, 17248, 25473, 25479, 25617, 25619, 25664, 28709
4948	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1047, 1088, 1097, 1200, 1208, 1250, 1252, 1282, 4386, 4909, 4929, 4932, 4934, 4946, 4951, 4953, 4970-4971, 5012, 5123, 5346, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 25664, 28709
4951	37, 259, 273, 277-278, 280, 284-285, 290, 297, 437, 500, 819, 833, 836, 850, 852, 855, 857, 866, 870-871, 878, 880, 912, 915, 1025-1027, 1040-1043, 1088, 1200, 1208, 1250-1252, 1283, 4386, 4929, 4932, 4946, 4948, 4953, 5123, 5346, 5347, 8229, 8482, 9025, 9044, 9049, 13121, 13488, 17584, 25617, 25619, 25664, 28709
4952	259, 273, 424, 500, 803, 850, 856, 862, 916, 1200, 1208, 1255, 4946, 5012, 5351, 13488, 17584
4953	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1088, 1097, 1252, 1254, 1281, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4970-4971, 5012, 5123, 5350, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 16804, 25473, 25479, 25617, 25619, 25664, 28709
4960	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 720, 819, 833, 838, 850, 864, 870-871, 875, 880, 905, 918, 1008, 1025-1027, 1046, 1089, 1097, 1127, 1200, 1208, 1252, 1256, 4386, 4929, 4934, 4946, 4971, 5104, 5123, 5142, 5352, 8229, 8482, 8612, 9025, 9030, 9056, 9238, 13121, 13488, 16804, 17248, 17584, 28709
4970	37, 259, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1252, 4909, 4934, 4946, 4948, 4953, 4971, 5012, 5123, 8229, 9030, 9044, 9049, 9061, 9066, 25473, 25479, 25617, 25619, 28709
4971	37, 256, 273, 277-278, 280, 284-285, 297, 367, 423, 437, 500, 737, 775, 813, 819, 836, 838, 850-852, 857, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1041-1043, 1047, 1088, 1112, 1122, 1200, 1208, 1252-1253, 1280, 4909, 4932, 4934, 4946, 4948, 4953, 4960, 4970, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 25664, 28709
4992	290, 896, 1027, 1041, 4386, 5123, 8482, 25617

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
5012	37, 273, 277-278, 280, 284-285, 297, 423-424, 437, 500, 803, 813, 819, 838, 850, 852, 856-857, 860-863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252, 1255, 4909, 4934, 4946, 4948, 4952-4953, 4970-4971, 5123, 5351, 8229, 9030, 9044, 9049, 9061, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
5026	930-932, 939, 942-943, 1390, 1399, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 1208, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
5028	930-932, 939, 942-943, 1390, 1399, 5026, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
5029	933-934, 944, 949, 1363-1364, 5045, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717, 37813
5031	935-936, 946, 1381, 1386, 1388, 5477, 5482, 5484, 9127, 13223, 25512
5033	937-938, 948, 950, 1370, 5046, 9142, 25514, 25524, 29620
5035	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5038-5039, 9122, 9124, 9131, 9135, 1208, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
5038	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
5039	930-932, 939, 942-943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038, 9122, 9124, 9131, 9135, 13218-13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
5045	933-934, 944, 949, 1363-1364, 5029, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717, 37813
5046	937-938, 948, 950, 1370, 5033, 9142, 25514, 25524, 29620
5104	420, 864, 1008, 1200, 1208, 4960, 8612, 9056, 13488, 16804, 17248, 17584
5123	290, 367, 423, 437, 819, 1027, 1041, 1047, 1140-1149, 1156, 1157, 1160, 1200, 1208, 1252, 4948, 5348, 8482, 13488
5142	420, 500, 864, 1046, 1089, 1127, 1200, 1208, 1256, 4960, 5352, 8612, 9056, 9238, 13488, 16804, 17248, 17584
5210	37, 437, 500, 819, 836, 850, 904, 1043, 1114-1115, 1200, 1208, 4932, 4946, 5211, 8229, 13488, 17584, 25480, 25619, 28709
5211	37, 367, 437, 500, 836, 903, 1114-1115, 4932, 5210, 8229, 25479, 28709
5346	37, 259, 273, 500, 819, 850, 852, 855, 870, 912, 1200, 1208, 1250, 1252, 1282, 4946, 4948, 4951, 8229, 9044, 13488, 17584, 28709
5347	808, 848-849, 855, 866, 872, 878, 880, 915, 1025, 1123-1125, 1131, 1154, 1158, 1200, 1208, 1251, 1283, 4951, 13488, 17584

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
5348	37, 259, 273, 275, 277-278, 280, 284-285, 297, 437, 500, 808, 819, 850, 858, 860-861, 863, 865, 871-872, 901-902, 923-924, 1051, 1140-1149, 1153-1158, 1160-1162, 1164, 1200, 1208, 1252, 1275, 4946, 8229, 13488, 17584, 28709
5349	813, 869, 875, 1148, 1200, 1208, 1253, 1280, 4909, 4971, 9061, 13488, 17584
5350	857, 920, 1026, 1155, 1200, 1208, 1254, 1281, 4953, 9049, 13488, 17584
5351	424, 856, 862, 867, 916, 1200, 1208, 1255, 4899, 4952, 5012, 9048, 12712, 13488, 17584
5352	420, 864, 1046, 1089, 1200, 1208, 1256, 4960, 5142, 8612, 9056, 9238, 13488, 16804, 17248, 17584
5353	901-902, 921-922, 1112, 1122, 1156-1157, 1200, 1208, 1257, 13488, 17584
5354	1129-1130, 1163, 1164, 1200, 1208, 1258, 13488, 17584
5460	933-934, 944, 949, 1363-1364, 5029, 5045, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717, 37813
5477	935-936, 1381, 1386, 1388, 5031, 5482, 5484, 9127, 13223, 25512
5482	935, 1381, 1386, 1388, 5031, 5477, 5484, 9127, 13223
5484	935-936, 946, 1381, 1386, 1388, 5031, 5477, 5482, 9127, 13223, 25512
5488	1388
8229	37, 256, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 720, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-866, 869-871, 874-875, 880, 897, 903-905, 912, 914-916, 920-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1097, 1100, 1112, 1114-1115, 1122, 1124, 1126, 1130-1132, 1137, 1140-1149, 1250-1255, 1257-1258, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5210-5211, 5346, 5348, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 16804, 17248, 25473, 25479, 25480, 25617, 25619, 25664, 28709
8482	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 895-897, 1009, 1025-1027, 1040-1043, 1047, 1088, 1112, 1122, 1139, 1200, 1208, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 4992, 5123, 8229, 9025, 9044, 9049, 9056, 13121, 13488, 17248, 17584, 25473, 25617, 25619, 25664, 28709
8492	300-301, 941, 1351, 4396, 16684
8612	37, 256, 420, 424, 437, 500, 720, 737, 775, 819, 850, 852, 857, 860-865, 1008, 1046, 1089, 1098, 1112, 1122, 1127, 1200, 1208, 1252, 1256, 4946, 4948, 4953, 4960, 5104, 5142, 5352, 8229, 9044, 9049, 9056, 9238, 13488, 16804, 17248, 17584, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
9025	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 891, 1009, 1025-1027, 1040-1043, 1088, 1112, 1122, 1126, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 5123, 8229, 8482, 9044, 9049, 9056, 13121, 17248, 25617, 25619, 25664, 28709
9026	834, 926, 951, 1362, 4930
9027	835, 927, 947, 1200, 1208, 4931, 13488, 17584, 21427
9030	37, 256, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 775, 813, 819, 838, 850, 852, 857, 860-865, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1112, 1122, 1200, 1208, 1252, 4909, 4934, 4946, 4948, 4953, 4960, 4970-4971, 5012, 5123, 8229, 9044, 9049, 9056, 9061, 9066, 13488, 17248, 17584, 25473, 25479, 25617, 25619, 28709
9044	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1047, 1088, 1097, 1153, 1200, 1208, 1250, 1252, 1282, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5346, 8229, 8482, 8612, 9025, 9030, 9049, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 25664, 28709
9048	867, 1200, 1208, 4899, 5351, 12712, 13488, 17584
9049	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1088, 1097, 1155, 1200, 1208, 1252, 1254, 1281, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5350, 8229, 8482, 8612, 9025, 9030, 9044, 9061, 9066, 13121, 13488, 16804, 17584, 25473, 25479, 25617, 25619, 25664, 28709
9056	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 720, 819, 833, 838, 850, 864, 870-871, 875, 880, 905, 918, 1008, 1025-1027, 1046, 1089, 1097, 1127, 1200, 1208, 1252, 1256, 4386, 4929, 4934, 4946, 4960, 4971, 5104, 5123, 5142, 5352, 8229, 8482, 8612, 9025, 9030, 9238, 13121, 13488, 16804, 17248, 17584, 28709
9061	37, 256, 259, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 737, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252-1254, 1280, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5349, 8229, 9030, 9044, 9049, 9066, 13488, 17584, 25473, 25479, 25617, 25619, 28709
9066	37, 259, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1200, 1208, 1252, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 8229, 9030, 9044, 9049, 9061, 13488, 17584, 25473, 25479, 25617, 25619, 28709
9122	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796

Table 571. IBM MQ for z/OS CCSID conversion support (continued)

CCSID	Converts to and from CCSIDS
9124	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
9125	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717, 37813
9127	935-936, 946, 1381, 1386, 1388, 5031, 5477, 5482, 5484, 13223, 25512
9131	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
9135	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
9142	937-938, 948, 950, 1370, 5033, 5046, 25514, 25524, 29620
9238	420, 500, 864, 1046, 1089, 1127, 1200, 1208, 1256, 4960, 5142, 5352, 8612, 9056, 13488, 16804, 17248, 17584
9555	933, 949, 1363-1364, 5029, 5045, 5460, 9125, 13221, 13651, 17317, 25525, 29621, 33717, 37813
12712	862, 867, 1148, 1156-1157, 1200, 1208, 4899, 5351, 9048, 13488, 17584
13121	37, 256, 273, 277-278, 280, 284-285, 290, 297, 367, 437, 500, 737, 775, 819, 833, 836, 850, 852, 855, 857, 860-865, 870-871, 891, 1009, 1025-1027, 1040-1043, 1088, 1112, 1122, 1126, 1200, 1208, 1252, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4960, 5123, 8229, 8482, 9025, 9044, 9049, 9056, 13488, 17248, 17584, 25617, 25619, 25664, 28709
13218	930-932, 939, 942-943, 1200, 1208, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13219, 13231, 13488, 17314, 17584, 25508, 25518, 29614, 33698-33700, 37796
13219	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218, 13231, 17314, 25508, 25518, 29614, 33698-33700, 37796
13221	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717, 37813
13223	935-936, 946, 1381, 1386, 1388, 5031, 5477, 5482, 5484, 9127, 25512
13231	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 17314, 25508, 25518, 29614, 33698-33700, 37796

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
13488	37, 256, 259, 273, 275, 277-278, 280, 282, 284-285, 290, 293, 297, 300-301, 367, 420, 423-424, 437, 500, 720, 737, 775, 803, 806, 808, 813, 819, 833-838, 848-852, 855-872, 874-875, 878, 880, 891, 895-897, 901-905, 912, 914-916, 918, 920-924, 927-928, 930, 932-933, 935, 937, 939, 941-944, 946-951, 1004, 1006, 1008-1019, 1025-1027, 1040-1043, 1046-1047, 1051, 1088-1089, 1097-1098, 1112, 1114-1115, 1122-1126, 1129-1133, 1137, 1140-1149, 1153-1160, 1164, 1166, 1200, 1208, 1250-1258, 1275-1277, 1280-1285, 1351, 1362-1364, 1370-1371, 1380-1381, 1385-1386, 1388, 1390, 1399, 4899, 4909, 4930, 4933, 4948, 4951-4952, 4960, 4971, 5012, 5039, 5104, 5123, 5142, 5210, 5346-5354, 8482, 8612, 9027, 9030, 9044, 9048-9049, 9056, 9061, 9066, 9238, 12712, 13121, 13218, 16684, 16804, 17248, 17584, 21427, 28709
13651	933, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 17317, 25525, 29621, 33717, 37813
16684	300-301, 941, 1200, 1208, 1351, 4396, 8492, 13488, 17584
16804	37, 256, 420, 424, 437, 500, 720, 737, 775, 819, 850, 852, 857, 860-865, 1008, 1046, 1089, 1098, 1112, 1122, 1127, 1200, 1208, 1252, 1256, 4946, 4948, 4953, 4960, 5104, 5142, 5352, 8229, 8612, 9044, 9049, 9056, 9238, 13488, 17248, 17584, 28709
17248	37, 256, 259, 273, 277-278, 280, 284-285, 290, 297, 420, 423-424, 500, 720, 819, 833, 838, 850, 864, 870-871, 875, 880, 905, 918, 1008, 1025-1027, 1046, 1089, 1097, 1127, 1200, 1208, 1252, 1256, 4386, 4929, 4934, 4946, 4960, 4971, 5104, 5123, 5142, 5352, 8229, 8482, 8612, 9025, 9030, 9056, 9238, 13121, 13488, 16804, 17584, 28709
17314	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 25508, 25518, 29614, 33698-33700, 37796
17317	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13651, 25510, 25520, 25525, 29616, 29621, 33717, 37813
17584	37, 256, 259, 273, 275, 277-278, 280, 282, 284-285, 290, 293, 297, 300-301, 367, 420, 423-424, 437, 500, 720, 737, 775, 803, 806, 808, 813, 819, 833-838, 848-852, 855-872, 874-875, 878, 880, 891, 895-897, 901-905, 912, 914-916, 918, 920-924, 927-928, 930, 932-933, 935, 937, 939, 941-944, 946-951, 1004, 1006, 1008-1019, 1025-1027, 1040-1043, 1046-1047, 1051, 1088-1089, 1097-1098, 1112, 1114-1115, 1122-1126, 1129-1133, 1137, 1140-1149, 1153-1160, 1164, 1166, 1200, 1208, 1250-1258, 1275-1277, 1280-1285, 1351, 1362-1364, 1370-1371, 1380-1381, 1385-1386, 1388, 1390, 1399, 4899, 4909, 4930, 4933, 4948, 4951-4952, 4960, 4971, 5012, 5039, 5104, 5123, 5142, 5210, 5346-5354, 8482, 8612, 9027, 9030, 9044, 9048-9049, 9056, 9061, 9066, 9238, 12712, 13121, 13218, 13488, 16684, 16804, 17248, 21427, 28709
21427	835, 927, 947, 1200, 1208, 4931, 9027, 13488, 17584
25473	37, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1252, 4386, 4909, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 8229, 8482, 9030, 9044, 9049, 9061, 9066, 25479, 25617, 25619, 28709

<i>Table 571. IBM MQ for z/OS CCSID conversion support (continued)</i>	
CCSID	Converts to and from CCSIDS
25479	37, 273, 277-278, 280, 284-285, 297, 423, 437, 500, 813, 819, 836, 838, 850, 852, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1115, 1252, 4909, 4932, 4934, 4946, 4948, 4953, 4970-4971, 5012, 5123, 5211, 8229, 9030, 9044, 9049, 9061, 9066, 25473, 25617, 25619, 28709
25480	37, 500, 904, 1114, 5210, 8229, 28709
25508	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25518, 29614, 33698-33700, 37796
25510	933-934, 949, 5029, 5045, 5460, 9125, 13221, 17317, 25525, 29621, 33717, 37813
25512	935-936, 946, 1381, 5031, 5477, 5484, 9127, 13223
25514	937-938, 950, 1370, 5033, 5046, 9142
25518	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 29614, 33698-33700, 37796
25520	933, 944, 949, 5029, 5045, 5460, 9125, 13221, 17317, 25525, 29616, 29621, 33717, 37813
25524	937, 948, 950, 1370, 5033, 5046, 9142, 29620
25525	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 29616, 29621, 33717, 37813
25617	37, 273, 277-278, 280, 284-285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 895-897, 903, 912, 916, 1025-1027, 1040-1043, 1088, 1252, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 4992, 5012, 5123, 8229, 8482, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 25473, 25479, 25619, 25664, 28709
25619	37, 273, 277-278, 280, 284-285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-861, 863, 869-871, 874-875, 880, 897, 903, 912, 916, 1025-1027, 1040-1043, 1088, 1114, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4970-4971, 5012, 5123, 5210, 8229, 8482, 9025, 9030, 9044, 9049, 9061, 9066, 13121, 25473, 25479, 25617, 25664, 28709
25664	37, 273, 277-278, 280, 284-285, 290, 297, 367, 500, 819, 833, 836, 850, 852, 855, 857, 870-871, 875, 891, 1025-1027, 1040-1043, 1088, 1126, 4386, 4929, 4932, 4946, 4948, 4951, 4953, 4971, 5123, 8229, 8482, 9025, 9044, 9049, 13121, 25617, 25619, 28709

Table 571. IBM MQ for z/OS CCSID conversion support (continued)	
CCSID	Converts to and from CCSIDS
28709	37, 256, 273, 275, 277-278, 280, 284-285, 290, 297, 367, 420, 423-424, 437, 500, 720, 737, 775, 813, 819, 833, 836, 838, 850, 852, 855, 857-858, 860-866, 869-871, 874-875, 880, 897, 903-905, 912, 914-916, 920-924, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1097, 1100, 1112, 1114-1115, 1122, 1124, 1126, 1130-1132, 1137, 1140-1149, 1200, 1208, 1250-1255, 1257-1258, 1275, 1280-1281, 1283, 4386, 4909, 4929, 4932, 4934, 4946, 4948, 4951, 4953, 4960, 4970-4971, 5012, 5123, 5210-5211, 5346, 5348, 8229, 8482, 8612, 9025, 9030, 9044, 9049, 9056, 9061, 9066, 13121, 13488, 16804, 17248, 17584, 25473, 25479, 25480, 25617, 25619, 25664
29614	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 33698-33700, 37796
29616	933, 944, 949, 5029, 5045, 5460, 9125, 13221, 17317, 25520, 25525, 29621, 33717, 37813
29620	937, 948, 950, 1370, 5033, 5046, 9142, 25524
29621	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 33717, 37813
33698	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33699-33700, 37796
33699	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698, 33700, 37796
33700	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33699, 37796
33717	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 37813
37796	930-932, 939, 942-943, 1390, 1399, 5026, 5028, 5035, 5038-5039, 9122, 9124, 9131, 9135, 13218-13219, 13231, 17314, 25508, 25518, 29614, 33698-33700
37813	933-934, 944, 949, 1363-1364, 5029, 5045, 5460, 9125, 9555, 13221, 13651, 17317, 25510, 25520, 25525, 29616, 29621, 33717

IBM i conversion support

A full list of CCSIDs, and conversions supported by IBM i, can be found in the appropriate IBM i publication.

The supported code pages are listed in [Supported CCSID mappings](#).

Unicode conversion support

Some platforms support the conversion of user data to or from Unicode encoding. The two forms of Unicode encoding supported are UCS-2 (CCSIDs 1200, 13488, and 17584) and UTF-8 (CCSID 1208).

The term *UCS-2* is often used interchangeably but incorrectly with *UTF-16*. UCS-2 is a fixed-width encoding where each character occupies 2 bytes. UTF-16 is variable-width encoding that is a superset of

UCS-2. In addition to the 2-byte UCS-2 characters, UTF-16 contains characters, known as surrogate pairs, that are 4 bytes in length. IBM MQ does not support surrogate pairs. The support for UTF-16 and UTF-8 in IBM MQ is therefore limited to those Unicode characters that can be encoded in UCS-2.

Note: IBM MQ does not support UCS-2 queue manager CCSIDs so message header data cannot be encoded in UCS-2.

IBM MQ AIX support for Unicode

On IBM MQ for AIX conversion to and from Unicode CCSIDs is supported for the CCSIDs in the following table.

037	273	278	280	284	285
297	423	437	500	813	819
850	852	856	857	858	860
861	865	867	869	875	878
880	901	902	912	915	916
920	923	924	932	933	935
937	938	939	942	943	948
949	950	954	964	970	1026
1046	1089	1129	1130	1131	1132
1133	1140	1141	1142	1143	1144
1145	1146	1147	1148	1149	1200
1153	1156	1157	1208	1250	1251
1253	1254	1258	1280	1281	1282
1283	1284	1285	1363	1364	1381
1383	1386	1388	4899	5026	5035
5050	5346	5347	5348	5349	5350
5351	5352	5353	5354	5488	9044
9048	9449	12712	13488	17584	33722

IBM MQ HP-UX support for Unicode

On IBM MQ for HP-UX conversion to and from Unicode CCSIDs is supported for the CCSIDs listed in the following table.

437	737	813	819	850	852
855	857	861	864	865	866
869	874	912	915	916	920
932	938	950	954	964	970
1051	1089	1140	1141	1142	1143
1144	1145	1146	1147	1148	1149
1200	1208	1250	1251	1252	1253
1254	1255	1256	1257	1258	1381
5050	5488	13488	33722		

IBM MQ for Windows, Solaris, and Linux support for Unicode

On IBM MQ for Windows, IBM MQ for Solaris, and IBM MQ for Linux conversion to, and from, Unicode CCSIDs is supported for the CCSIDs in the following table.

037	277	278	280	284	285
290	297	300	301	420	424
437	500	813	819	833	835
836	837	838	850	852	855
856	857	858	860	861	862
863	864	865	866	867	868
869	870	871	874	875	878
880	891	897	901	902	903
904	912	913 (5)	915	916	918
920	921	922	923	924	927
928	930	931 (1)	932 (2)	933	935
937	938 (3)	939	941	942	943
947	948	949	950	951	954 (4)
964	970	1006	1025	1026	1027
1040	1041	1042	1043	1046	1047
1051	1088	1089	1097	1098	1112
1114	1115	1122	1123	1124	1129
1130	1132	1133	1140	1141	1142
1143	1144	1145	1146	1147	1148
1149	1153	1156	1157	1200	1208
1250	1251	1252	1253	1254	1255
1256	1257	1258	1275	1280	1281
1282	1283	1363	1364	1380	1381
1383	1386	1388	4899	5050	5346
5347	5348	5349	5350	5351	5352
5353	5354	5488 (5)	9044	9048	9449
12712	13488	17584	33722 (4)		

Notes:

1. 931 uses 939 for conversion.
2. 932 uses 942 for conversion.
3. 938 uses 948 for conversion.
4. 954 and 33722 use 5050 for conversion.
5. On Windows, Linux, and Solaris only.

IBM i support for Unicode

For details on UNICODE support refer to the appropriate IBM i publication relating to your operating system.

IBM MQ for z/OS support for Unicode

On IBM MQ for z/OS conversion to and from the Unicode CCSIDs is supported for the following CCSIDs:

37	256	259	273	275	277
278	280	282	284	285	290
293	297	300	301	367	420
423	424	437	500	720	737
775	803	806	808	813	819
833	834	835	836	837	838
848	849	850	851	852	855
856	857	858	859	860	861
862	863	864	865	866	867
868	869	870	871	872	874
875	878	880	891	895	896
897	901	902	903	904	905
912	914	915	916	918	920
921	922	923	924	927	928
930	932	933	935	937	939
941	942	943	944	946	947
948	949	950	951	1004	1006
1008	1009	1010	1011	1012	1013
1014	1015	1016	1017	1018	1019
1025	1026	1027	1040	1041	1042
1043	1046	1047	1051	1088	1089
1097	1098	1112	1114	1115	1122
1123	1124	1125	1126	1129	1130
1131	1132	1133	1137	1140	1141
1142	1143	1144	1145	1146	1147
1148	1149	1153	1154	1155	1156
1157	1158	1159	1160	1161	1162
1164	1200	1208	1250	1251	1252
1253	1254	1255	1256	1257	1258
1275	1276	1277	1280	1281	1282
1283	1284	1285	1351	1362	1363
1364	1370	1371	1380	1381	1385
1386	1388	1390	1399	4899	4909

4930	4933	4948	4951	4952	4960
4971	5012	5039	5104	5123	5142
5210	5346	5347	5348	5349	5350
5351	5352	5353	5354	5488	8482
8612	9027	9030	9044	9048	9049
9056	9061	9066	9238	9449	1166
12712	13121	13218	13488	16684	16804
17248	17584	21427	28709		

Coding standards on 64-bit platforms

Use this information to learn about coding standards on 64-bit platforms and the preferred data types.

Preferred data types

These types never change size and are available on both 32-bit and 64-bit IBM MQ platforms:

Name	Length
MQLONG	4 bytes
MQULONG	4 bytes
MQINT32	4 bytes
MQUINT32	4 bytes
MQINT64	8 bytes
MQUINT64	8 bytes

Standard data types

Learn about standard data types on 32-bit UNIX, 64-bit UNIX, and 64-bit Windows applications.

32-bit UNIX applications

This section is included for comparison and is based on Solaris. Any differences with other UNIX platforms are noted:

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	16 bytes
Note that on AIX and Linux PPC a long double is 8 bytes.	
pointer	4 bytes
ptrdiff_t	4 bytes

Name	Length
size_t	4 bytes
time_t	4 bytes
clock_t	4 bytes
wchar_t	4 bytes

Note that on AIX a wchar_t is 2 bytes.

64-bit UNIX applications

This section is based on Solaris. Any differences with other UNIX platforms are noted:

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
long double	16 bytes

Note that on AIX and Linux PPC a long double is 8 bytes.

pointer	8 bytes
ptrdiff_t	8 bytes
size_t	8 bytes
time_t	8 bytes
clock_t	8 bytes

Note that on the other UNIX platforms a clock_t is 4 bytes.

wchar_t	4 bytes
---------	---------

Note that on AIX a wchar_t is 2 bytes.

Windows 64-bit applications

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

Name	Length
pointer	8 bytes
	Note that all pointers are 8 bytes.
ptrdiff_t	8 bytes
size_t	8 bytes
time_t	8 bytes
clock_t	4 bytes
wchar_t	2 bytes
WORD	2 bytes
DWORD	4 bytes
HANDLE	8 bytes
HFILE	4 bytes

Coding considerations on Windows

HANDLE hf;

Use

```
hf = CreateFile((LPCTSTR) FileName,
                Access,
                ShareMode,
                xihSecAttsNTRestrict,
                Create,
                AttrAndFlags,
                NULL);
```

Do not use

```
HFILE hf;
hf = (HFILE) CreateFile((LPCTSTR) FileName,
                        Access,
                        ShareMode,
                        xihSecAttsNTRestrict,
                        Create,
                        AttrAndFlags,
                        NULL);
```

as this produces an error.

size_t len fgets

Use

```
size_t len
while (fgets(string1, (int) len, fp) != NULL)
len = strlen(buffer);
```

Do not use

```
int len;

while (fgets(string1, len, fp) != NULL)
len = strlen(buffer);
```

printf

Use

```
printf("My struc pointer: %p", pMyStruc);
```

Do not use

```
printf("My struc pointer: %x", pMyStruc);
```

If you need hexadecimal output, you have to print the upper and lower 4 bytes separately.

char *ptr

Use

```
char * ptr1;
char * ptr2;
size_t buflen;

buflen = ptr2 - ptr1;
```

Do not use

```
char *ptr1;
char *ptr2;
UINT32 buflen;

buflen = ptr2 - ptr1;
```

alignBytes

Use

```
alignBytes = (unsigned short) ((size_t) address % 16);
```

Do not use

```
void *address;
unsigned short alignBytes;

alignBytes = (unsigned short) ((UINT32) address % 16);
```

len

Use

```
len = (UINT32) ((char *) address2 - (char *) address1);
```

Do not use

```
void *address1;
void *address2;
UINT32 len;

len = (UINT32) ((char *) address2 - (char *) address1);
```

sscanf

Use

```
MQLONG SBCSprt;

sscanf(line, "%d", &SBCSprt);
```

Do not use

```
MQLONG SBCSprt;

sscanf(line, "%1d", &SBCSprt);
```

%1d tries to put an 8-byte type into a 4-byte type; only use %1 if you are dealing with an actual long data type. MQLONG, UINT32 and INT32 are defined to be four bytes, the same as an int on all IBM MQ platforms:

Application programming for IBM i

Use the following topics to help you develop applications for IBM i:

- [“Data type descriptions” on page 958](#)
- [“Function calls” on page 1206](#)
- [“Attributes of objects” on page 1324](#)
- [“Applications” on page 1369](#)
- [“Return codes for IBM i \(ILE RPG\)” on page 1381](#)
- [“Rules for validating MQI options for IBM i \(ILE RPG\)” on page 1382](#)
- [“Machine encodings on IBM i” on page 1385](#)
- [“Report options and message flags” on page 1388](#)

Compatibility mode for RPG applications on IBM i

This mode of operation was needed for applications that are written before MQSeries Version 5.1. Subsequent versions of the product provide a compatible runtime environment for these applications, even though the copybooks needed for compiling them were removed in IBM WebSphere MQ 6.0. Although dynamic linkage (compatibility mode) is provided in library QMQM for versions of the product after MQSeries Version 5.1, applications that use this compatibility mode of operation, need to be recompiled to use the static bound MQ calls that are provided by the LIBMQM and LIBMQM_R service programs, instead.

Related tasks

[Developing applications](#)

Data type descriptions

This collection of topics provides descriptions of data types used in IBM i programming.

Conventions used in the description of data types

For each elementary data type, this information gives a description of its usage, in a form that is independent of the programming language. This is followed by typical declarations in the ILE version of the RPG programming language. The definitions of elementary data types are included here to provide consistency. RPG uses 'D' specifications where working fields can be declared using whatever attributes you need. You can, however, do this in the calculation specifications where the field is used.

To use the elementary data types, you create:

- A /COPY member containing all the data types, or
- An external data structure (PF) containing all the data types. You then need to specify your working fields with attributes 'LIKE' the appropriate data type field.

The benefits of the second option are that the definitions can be used as a 'FIELD REFERENCE FILE' for other IBM i objects. If an IBM MQ data type definition changes, it is a relatively simple matter to re-create these objects.

Elementary data types

All of the other data types described in this section equate either directly to these elementary data types, or to aggregates of these elementary data types (arrays or structures).

<i>Table 572. Elementary data types</i>	
Data type	Representation
MQBOOL	10-digit signed integer
MQBYTE	1-byte alphanumeric field
MQBYTE16	16-byte alphanumeric field
MQBYTE24	24-byte alphanumeric field
MQBYTE32	32-byte alphanumeric field
MQBYTE64	64-byte alphanumeric field
MQCHAR	1-byte alphanumeric field
MQCHAR4	4-byte alphanumeric field
MQCHAR8	8-byte alphanumeric field
MQCHAR12	12-byte alphanumeric field
MQCHAR16	16-byte alphanumeric field
MQCHAR20	20-byte alphanumeric field
MQCHAR28	28-byte alphanumeric field
MQCHAR32	32-byte alphanumeric field
MQCHAR48	48-byte alphanumeric field
MQCHAR64	64-byte alphanumeric field
MQCHAR128	128-byte alphanumeric field
MQCHAR256	256-byte alphanumeric field
MQFLOAT32	4-byte floating-point number
MQFLOAT64	8-byte floating-point number
MQHCONFIG	Configuration handle
MQHCONN	10-digit signed integer
MQHMSG	Message handle that gives access to a message
MQHOBJ	10-digit signed integer
MQINT8	8-bit signed integer
MQINT16	16-bit signed integer
MQINT32	32-bit signed integer
MQINT64	64-bit signed integer
MQLONG	32-bit signed integer
MQPID	Process identifier
MQPTR	Pointer

Table 572. Elementary data types (continued)

Data type	Representation
MQTID	Thread identifier
MQUINT8	8-bit unsigned integer
MQUINT16	16-bit unsigned integer
MQUINT32	32-bit unsigned integer
MQUINT64	64-bit unsigned integer
MQULONG	32-bit unsigned integer
PMQACH	Pointer to a data structure of type MQACH
PMQAIR	Pointer to a data structure of type MQAIR
PMQAXC	Pointer to a data structure of type MQAXC
PMAXP	Pointer to a data structure of type MAXP
PMQBMHO	Pointer to a data structure of type MQBMHO
PMQBO	Pointer to a data structure of type MQBO
PMQBOOL	Pointer to data of type MQBOOL
PMQBYTE	Pointer to data of type MQBYTE
PMQBYTEn	Pointer to data of type MQBYTEn
PMQCBC	Pointer to a data structure of type MQCBC
PMQCBD	Pointer to a data structure of type MQCBD
PMQCHAR	Pointer to a data structure of type MQCHAR
PMQCHARV	Pointer to a data structure of type MQCHARV
PMQCHARn	Pointer to data of type MQCHARn
PMQCIH	Pointer to a data structure of type MQCIH
PMQCMHO	Pointer to a data structure of type MQCMHO
PMQCNO	Pointer to a data structure of type MQCNO
PMQCSP	Pointer to a data structure of type MQCSP
PMQCTLO	Pointer to a data structure of type MQCTLO
PMQDH	Pointer to a data structure of type MQDH
PMQDHO	Pointer to a data structure of type MQDHO
PMQDLH	Pointer to a data structure of type MQDLH
PMQDMHO	Pointer to a data structure of type MQDMHO
PMQDMPO	Pointer to a data structure of type MQDMPO
PMQEPH	Pointer to a data structure of type MQEPH
PMQFLOAT32	Pointer to data of type MQFLOAT32
PMQFLOAT64	Pointer to data of type MQFLOAT64
PMQFUNC	Pointer to a function

Table 572. Elementary data types (continued)

Data type	Representation
PMQGM0	Pointer to a data structure of type MQGM0
PMQHCONFIG	Pointer to data of type MQHCONFIG
PMQHCONN	Pointer to data of type MQHCONN
PMQHMSG	Pointer to data of type MQHMSG
PMQHOBj	Pointer to data of type MQHOBj
PMQIIH	Pointer to a data structure of type MQIIH
PMQIMPO	Pointer to a data structure of type MQIMPO
PMQINT8	Pointer to data of type MQINT8
PMQINT16	Pointer to data of type MQINT16
PMQINT32	Pointer to data of type MQINT32
PMQINT64	Pointer to data of type MQINT64
PMQLONG	Pointer to data of type MQLONG
PMQMD	Pointer to a data structure of type MQMD
PMQMDE	Pointer to a data structure of type MQMDE
PMQMD1	Pointer to a data structure of type MQMD1
PMQMD2	Pointer to a data structure of type MQMD2
PMQMHBO	Pointer to a data structure of type MQMHBO
PMQOD	Pointer to a data structure of type MQOD
PMQOR	Pointer to a data structure of type MQOR
PMQPD	Pointer to a data structure of type MQPD
PMQPID	Pointer to a process identifier MQPID
PMQPMO	Pointer to a data structure of type MQPMO
PMQPTR	Pointer to data of type MQPTR
PMQRFH	Pointer to a data structure of type MQRFH
PMQRFH2	Pointer to a data structure of type MQRFH2
PMQRMH	Pointer to a data structure of type MQRMH
PMQRR	Pointer to a data structure of type MQRR
PMQSCO	Pointer to a data structure of type MQSCO
PMQSD	Pointer to a data structure of type MQSD
PMQSMPO	Pointer to a data structure of type MQSMPO
PMQSRO	Pointer to a data structure of type MQSRO
PMQSTS	Pointer to a data structure of type MQSTS
PMQTID	Pointer to a thread identifier MQTID
PMQTM	Pointer to a data structure of type MQTM

Table 572. Elementary data types (continued)

Data type	Representation
PMQPMC2	Pointer to a data structure of type MQPMC2
PMQUINT8	Pointer to data of type MQUINT8
PMQUINT16	Pointer to data of type MQUINT16
PMQUINT32	Pointer to data of type MQUINT32
PMQUINT64	Pointer to data of type MQUINT64
PMQULONG	Pointer to data of type MQULONG
PMQVOID	Pointer
PMQWIH	Pointer to a data structure of type MQWIH
PMQXQH	Pointer to a data structure of type MQXQH

MQBOOL

The MQBOOL data type represents a boolean value. The value 0 represents false. Any other value represents true.

An MQBOOL must be aligned as for the MQLONG data type.

MQBYTE - Byte

The MQBYTE data type represents a single byte of data.

No particular interpretation is placed on the byte—it is treated as a string of bits, and not as a binary number or character. No special alignment is required.

An array of MQBYTE is sometimes used to represent an area of main storage with a nature that is not known to the queue manager. For example, the area might contain application message data or a structure. The boundary alignment of this area must be compatible with the nature of the data contained within it.

MQBYTEN - String of *n* bytes

Each MQBYTEN data type represents a string of *n* bytes.

Where *n* can take one of the following values:

- 16, 24, 32, or 64.

Each byte is described by the MQBYTE data type. No special alignment is required.

If the data in the string is shorter than the defined length of the string, the data must be padded with nulls to fill the string.

When the queue manager returns byte strings to the application (for example, on the MQGET call), the queue manager always pads with nulls to the defined length of the string.

Constants are available that define the lengths of byte string fields.

MQCHAR - character

The MQCHAR data type represents a single character.

The coded character set identifier of the character is that of the queue manager (see the *CodedCharSetId* attribute in topic [CodedCharSetId](#)). No special alignment is required.

Note: Application message data specified on the MQGET, MQPUT, and MQPUT1 calls is described by the MQBYTE data type, not the MQCHAR data type.

MQCHARn - String of n characters

Each MQCHARn data type represents a string of n characters.

Where n can take one of the following values:

- 4, 8, 12, 16, 20, 28, 32, 48, 64, 128, or 256

Each character is described by the MQCHAR data type. No special alignment is required.

If the data in the string is shorter than the defined length of the string, the data must be padded with blanks to fill the string. In some cases a null character can be used to end the string prematurely, instead of padding with blanks; the null character and characters following it are treated as blanks, up to the defined length of the string. The places where a null can be used are identified in the call and data type descriptions.

When the queue manager returns character strings to the application (for example, on the MQGET call), the queue manager always pads with blanks to the defined length of the string; the queue manager does not use the null character to delimit the string.

Constants are available that define the lengths of character string fields.

MQFLOAT32

The MQFLOAT32 data type is a 32-bit floating-point number represented using the standard IEEE floating-point format.

An MQFLOAT32 must be aligned on a 4-byte boundary.

MQFLOAT64

The MQFLOAT64 data type is a 64-bit floating-point number represented using the standard IEEE floating-point format.

An MQFLOAT64 must be aligned on an 8-byte boundary.

MQHCONFIG - configuration handle

The MQHCONFIG data type represents a configuration handle, that is, the component that is being configured for a particular installable service. A configuration handle must be aligned on its natural boundary.

Note: Applications must test variables of this type for equality only.

MQHCONN - Connection handle

The MQHCONN data type represents a connection handle, that is, the connection to a particular queue manager.

A connection handle must be aligned on its natural boundary.

Note: Applications must test variables of this type for equality only.

MQHMSG - Message handle

The MQHMSG data type represents a message handle that gives access to a message.

A message handle must be aligned on an 8-byte boundary.

Note: Applications must test variables of this type for equality only.

MQHOBJ - Object handle

The MQHOBJ data type represents an object handle that gives access to an object.

An object handle must be aligned on its natural boundary.

Note: Applications must test variables of this type for equality only.

MQINT8 - 8-bit signed integer

The MQINT8 data type is an 8-bit signed integer that can take any value in the range -128 to +127, unless otherwise restricted by the context.

MQINT16 - 16-bit signed integer

The MQINT16 data type is a 16-bit signed integer that can take any value in the range -32 768 to +32 767, unless otherwise restricted by the context.

An MQINT16 must be aligned on a 2-byte boundary.

MQINT32 - 32-bit integer

The MQINT32 data type is a 32-bit signed integer.

It is equivalent to MQLONG.

MQINT64 - 64-bit integer

The MQINT64 data type is a 64-bit signed integer that can take any value in the range -9 223 372 036 854 775 808 through +9 223 372 036 854 775 807, unless otherwise restricted by the context.

For COBOL, the valid range is limited to -999 999 999 999 999 999 through +999 999 999 999 999 999. An MQINT64 should be aligned on an 8-byte boundary.

MQLONG - Long integer

The MQLONG data type is a 32-bit signed binary integer that can take any value in the range -2 147 483 648 through +2 147 483 647, unless otherwise restricted by the context, aligned on its natural boundary.

MQPID - process identifier

The IBM MQ process identifier.

This is the same identifier used in IBM MQ trace and FFST dumps, but might be different from the operating system process identifier.

MQPTR - pointer

The MQPTR data type is the address of data of any type. A pointer must be aligned on its natural boundary; this is a 16-byte boundary on IBM i.

Some programming languages support typed pointers; the MQI also uses these in a few cases.

MQTID - thread identifier

The MQ thread identifier.

This is the same identifier used in MQ trace and FFST dumps, but might be different from the operating system thread identifier.

MQUINT8 - 8-bit unsigned integer

The MQUINT8 data type is an 8-bit unsigned integer that can take any value in the range 0 to +255, unless otherwise restricted by the context.

MQUINT16 - 16-bit unsigned integer

The MQUINT16 data type is a 16-bit unsigned integer that can take any value in the range 0 through +65 535, unless otherwise restricted by the context.

An MQUINT16 must be aligned on a 2-byte boundary.

MQUINT32 - 32-bit unsigned integer

The MQUINT32 data type is a 32-bit unsigned integer. It is equivalent to MQULONG.

MQUINT64 - 64-bit unsigned integer

The MQUINT64 data type is a 64-bit unsigned integer that can take any value in the range 0 through +18 446 744 073 709 551 615 unless otherwise restricted by the context.

For COBOL, the valid range is limited to 0 through +999 999 999 999 999 999. An MQUINT64 should be aligned on a 8-byte boundary.

MQULONG - 32-bit unsigned integer

The MQULONG data type is a 32-bit unsigned binary integer that can take any value in the range 0 through +4 294 967 294, unless otherwise restricted by the context.

An MQULONG must be aligned on a 4-byte boundary.

PMQACH - pointer to a data structure of type MQACH

A pointer to a data structure of type MQACH.

PMQAIR - pointer to a data structure of type MQAIR

A pointer to a data structure of type MQAIR.

PMQAXC - pointer to a data structure of type MQAXC

A pointer to a data structure of type MQAXC.

PMQAXP - pointer to a data structure of type MQAXP

A pointer to a data structure of type MQAXP.

PMQBMHO - pointer to a data structure of type MQBMHO

A pointer to a data structure of type MQBMHO.

PMQBO - pointer to a data structure of type MQBO

A pointer to a data structure of type MQBO.

PMQBOOL - pointer to data of type MQBOOL

A pointer to data of type MQBOOL.

A pointer to data of type MQBOOL.

PMQBYTE - pointer to a data type of MQBYTE

A pointer to a data type of MQBYTE.

PMQBYTEn - pointer to a data structure of type MQBYTEn

A pointer to a data structure of type MQBYTEn, where n can be 8, 12, 16, 24, 32, 40, 48 or 128.

PMQCBC - pointer to a data structure of type MQCBC

A pointer to a data structure of type MQCBC.

PMQCBD - pointer to a data structure of type MQCBD

A pointer to a data structure of type MQCBD.

PMQCHAR - pointer to data of type MQCHAR

A pointer to data of type MQCHAR.

PMQCHARV - pointer to a data structure of type MQCHARV

A pointer to a data structure of type MQCHARV.

PMQCHARn - pointer to a data type of MQCHARn

A pointer to a data type of MQCHARn, where n can be 4, 8, 12, 20, 28, 32, 64, 128, 256, 264.

PMQCIH - pointer to a data structure of type of MQCIH

A pointer to a data structure of type of MQCIH.

PMQCMHO - pointer to a data structure of type MQCMHO

A pointer to a data structure of type MQCMHO.

PMQCNO - pointer to a data structure of type of MQCNO

A pointer to a data structure of type of MQCNO.

PMQCSP - pointer to a data structure of type MQCSP

A pointer to a data structure of type MQCSP.

PMQCTLO - pointer to a data structure of type MQCTLO

A pointer to a data structure of type MQCTLO.

PMQDH - pointer to a data structure of type MQDH

A pointer to a data structure of type MQDH.

PMQDHO - pointer to a data structure of type MQDHO

A pointer to a data structure of type MQDHO.

PMQDLH - pointer to a data structure of type of MQDLH

A pointer to a data structure of type of MQDLH.

PMQDMHO - pointer to a data structure of type MQDMHO

A pointer to a data structure of type MQDMHO.

PMQDMPO - pointer to a data structure of type MQDMPO

A pointer to a data structure of type MQDMPO.

A pointer to a data structure of type MQDMPO.

PMQEPPH - pointer to a data structure of type MQEPPH

A pointer to a data structure of type MQEPPH.

PMQFLOAT32 - pointer to data of type MQFLOAT32

A pointer to data of type MQFLOAT32.

PMQFLOAT64 - pointer to data of type MQFLOAT64

A pointer to data of type MQFLOAT64.

PMQFUNC - pointer to a function

A pointer to a function.

PMQGMPO - pointer to a data structure of type MQGMPO

A pointer to a data structure of type MQGMPO.

PMQHCONFIG - pointer to a data type of MQHCONFIG

A pointer to a data type of MQHCONFIG.

PMQHCONN - pointer to a data type of MQHCONN

A pointer to a data type of MQHCONN.

PMQHMSG - pointer to a data type of MQHMSG

A pointer to a data type of MQHMSG.

PMQHOBJS - pointer to data of type MQHOBJS

A pointer to data of type MQHOBJS.

PMQIIH - pointer to a data structure of type MQIIH

A pointer to a data structure of type MQIIH.

PMQIMPO - pointer to a data structure of type MQIMPO

A pointer to a data structure of type MQIMPO.

PMQINT8 - pointer to data of type MQINT8

A pointer to data of type MQINT8.

PMQINT16 - pointer to data of type MQINT16

A pointer to data of type MQINT16.

PMQINT32 - Pointer to data of type MQINT32

The PMQINT32 data type is a pointer to data of type MQINT32. It is equivalent to PMQLONG.

PMQINT64 - Pointer to data of type MQINT64

The PMQINT64 data type is a pointer to data of type MQINT64.

PMQLONG - pointer to data of type MQLONG

A pointer to data of type MQLONG.

PMQMD - pointer to structure of type MQMD

A pointer to structure of type MQMD.

PMQMDE - pointer to a data structure of type MQMDE

A pointer to a data structure of type MQMDE.

PMQMDI - pointer to a data structure of type MQMDI

A pointer to a data structure of type MQMDI.

PMQMD2 - pointer to a data structure of type MQMD2

A pointer to a data structure of type MQMD2

PMQMHBO - pointer to a data structure of type MQMHBO

A pointer to a data structure of type MQMHBO.

PMQOD - pointer to a data structure of type MQOD

A pointer to a data structure of type MQOD.

PMQOR - pointer to a data structure of type MQOR

A pointer to a data structure of type MQOR.

PMQPD - pointer to a data structure of type MQPD

A pointer to a data structure of type MQPD.

PMQPID - pointer to a process identifier

A pointer to a process identifier.

MQPMO - pointer to a data structure of type MQPMO

A pointer to a data structure of type MQPMO.

MQPTR - pointer to data of type MQPTR

A pointer to data of type MQPTR.

MQRFH - pointer to a data structure of type MQRFH

A pointer to a data structure of type MQRFH.

MQRFH2 - pointer to a data structure of type MQRFH2

A pointer to a data structure of type MQRFH2.

.

MQRMH - pointer to a data structure of type MQRMH

A pointer to a data structure of type MQRMH.

MQRR - pointer to a data structure of type MQRR

A pointer to a data structure of type MQRR.

MQSCO - pointer to a data structure of type MQSCO

A pointer to a data structure of type MQSCO.

.

MQSD - pointer to a data structure of type MQSD

A pointer to a data structure of type MQSD.

PMQSMPO - pointer to a data structure of type MQSMPO

A pointer to a data structure of type MQSMPO.

PMQSRO - pointer to a data structure of type MQSRO

A pointer to a data structure of type MQSRO.

PMQSTS - pointer to a data structure of type MQSTS

A pointer to a data structure of type MQSTS.

PMQTID - pointer to a data structure of type MQTID

A pointer to a data structure of type MQTID.

PMQTM - pointer to a data structure of type MQTM

A pointer to a data structure of type MQTM.

PMQTM2 - pointer to a data structure of type MQTM2

A pointer to a data structure of type MQTM2.

PMQUINT8 - pointer to data of type MQUINT8

A pointer to data of type MQUINT8.

PMQUINT16 - pointer to data of type MQUINT16

A pointer to data of type MQUINT16.

PMQUINT32 - Pointer to data of type MQUINT32

The PMQUINT32 data type is a pointer to data of type MQUINT32. It is equivalent to PMQULONG.

PMQUINT64 - Pointer to data of type MQUINT64

The PMQUINT64 data type is a pointer to data of type MQUINT64.

PMQULONG - pointer to data of type MQULONG

A pointer to data of type MQULONG.

PMQVOID - pointer

A pointer.

PMQWIH - pointer to a data structure of type MQWIH

A pointer to a data structure of type MQWIH.

PMQXQH - pointer to a data structure of type MQXQH

A pointer to a data structure of type MQXQH.

Language considerations

This topic contains information to help you use the MQI from the RPG programming language.

Some of these language considerations are:

- [“COPY files” on page 971](#)
- [“Calls” on page 973](#)
- [“Call parameters” on page 973](#)
- [“Structures” on page 973](#)
- [“Named constants” on page 973](#)
- [“MQI procedures” on page 974](#)
- [“Threading considerations” on page 974](#)
- [“Commitment control” on page 974](#)
- [“Coding the bound calls” on page 974](#)
- [“Notational conventions” on page 976](#)

COPY files

Various COPY files are provided to assist with the writing of RPG application programs that use message queuing. There are three sets of COPY files:

- COPY files with names ending with the letter *G* are for use with programs that use static linkage. These files are initialized with the exceptions stated in [“Structures” on page 973](#).
- COPY files with names ending with the letter *H* are for use with programs that use static linkage, but are **not** initialized.
- COPY files with names ending with the letter *R* are for use with programs that use dynamic linkage. These files are initialized with the exceptions stated in [“Structures” on page 973](#).

The COPY files reside in QRPGLSRC in the QMQM library.

For each set of COPY files, there are two files containing named constants, and one file for each of the structures. The COPY files are summarized in [Table 573 on page 971](#).

<i>Table 573. RPG COPY files</i>			
File name (static linkage, initialized, CMQ*G)	File name (static linkage, not initialized, CMQ*H)	File name (dynamic linkage, initialized, CMQ*R)	Contents
CMQBOG	CMQBOH	-	Begin options structure
CMQCDG	CMQCDH	CMQCDR	Channel definition structure
CMQCFBFG	CMQCFBFH	-	PCF bit filter parameter
CMQCFG	-	-	Constants for PCF and events
CMQCFBSG	CMQCFBSH	-	PCF byte string

Table 573. RPG COPY files (continued)

File name (static linkage, initialized, CMQ*G)	File name (static linkage, not initialized, CMQ*H)	File name (dynamic linkage, initialized, CMQ*R)	Contents
CMQCFGRG	CMQCFGRH	-	PCF group parameter
CMQCFIFG	CMQCFIFH	-	PCF integer filter parameter
CMQCFHG	CMQCFHH	-	PCF header
CMQCFILG	CMQCFILH	-	PCF integer list parameter structure
CMQCFING	CMQCFINH	-	PCF integer parameter structure
CMQCFSFG	CMQCFSFH	-	PCF string filter parameter
CMQCFSLG	CMQCFSLH	-	PCF string list parameter structure
CMQCFSTG	CMQCFSTH	-	PCF string parameter structure
CMQCFXLG	CMQCFXLH	-	PCF short name for CFIL64
CMQCFXNG	CMQCFXNH	-	PCF short name for CFIN64
CMQCIHG	CMQCIHH	-	CICS information header structure
CMQCNORG	CMQCNOH	-	Connect options structure
CMQCSPG	CMQCSPH	-	Security parameters
CMQCXPG	CMQCXPH	CMQCXPR	Channel exit parameter structure
CMQDHG	CMQDHH	CMQDHR	Distribution header structure
CMQDLHG	CMQDLHH	CMQDLHR	Dead letter header structure
CMQDXPG	CMQDXPH	CMQDXPR	Data conversion exit parameter structure
CMQEPHG	CMQEPHH	-	Embedded PCF header structure
CMQG	-	CMQR	Named constants for main MQI
CMQGMORG	CMQGMOH	CMQGMOR	Get message options structure
CMQIIHG	CMQIIHH	CMQIIHR	IMS information header structure
CMQMDEG	CMQMDEH	CMQMDER	Message descriptor extension structure
CMQMDG	CMQMDH	CMQMDR	Message descriptor structure
CMQMD1G	CMQMD1H	CMQMD1R	Message descriptor structure version 1
CMQMD2G	CMQMD2H	-	Message descriptor structure version 2
CMQODG	CMQODH	CMQODR	Object descriptor structure
CMQORG	CMQORH	CMQORR	Object record structure
CMQPMORG	CMQPMOH	CMQPMOR	Put message options structure
CMQPSG	-	-	Constants for publish/subscribe
CMQRFHG	CMQRFHH	-	Rules and formatting header structure
CMQRFH2G	CMQRFH2H	-	Rules and formatting header 2 structure
CMQRMHG	CMQRMHH	CMQRMHR	Reference message header structure

Table 573. RPG COPY files (continued)

File name (static linkage, initialized, CMQ*G)	File name (static linkage, not initialized, CMQ*H)	File name (dynamic linkage, initialized, CMQ*R)	Contents
CMQRRG	CMQRRH	CMQRRR	Response record structure
CMQTMCG	CMQTMCH	CMQTMCR	Trigger message structure (character format)
CMQTMCG	CMQTMCH	CMQTMCR	Trigger message structure (character format) version 2
CMQTMG	CMQTMH	CMQTMR	Trigger message structure
CMQWIHG	CMQWIHH	-	Work information header structure
CMQXG	-	CMQXR	Named constants for data conversion exit
CMQXQHG	CMQXQHH	CMQXQHR	Transmission queue header structure

Calls

Calls are described using their individual names.

Call parameters

Some parameters passed to the MQI can have more than one concurrent function. This is because the integer value passed is often tested on the setting of individual bits within the field, and not on its total value. This allows you to 'add' several functions together and pass them as a single parameter.

Structures

All IBM MQ structures are defined with initial values for the fields, with the following exceptions:

- Any structure with a suffix of H.
- MQTMC
- MQTMC2

These initial values are defined in the relevant table for each structure.

The structure declarations do not contain DS statements. This allows the application to declare either a single data structure or a multiple-occurrence data structure, by coding the DS statement and then using the /COPY statement to copy in the remainder of the declaration:

```
D*.1.....2.....3.....4.....5.....6.....7
D* Declare an MQMD data structure with 5 occurrences
DMYMD      DS              5
D/COPY CMQMDR
```

Named constants

There are many integer and character values that provide data interchange between your application program and the queue manager. To facilitate a more readable and consistent approach to using these values, named constants are defined for them. You can use these named constants and not the values they represent, as this improves the readability of the program source code.

When the COPY file CMQG is included in a program to define the constants, the RPG compiler will issue many severity-zero messages for the constants that are not used by the program; these messages are benign, and can safely be ignored.

MQI procedures

When using the ILE bound calls, you must bind to the MQI procedures when you create your program. These procedures are exported from the following service programs as appropriate:

QMQM/AMQZSTUB

This service program provides compatibility bindings for applications written before MQSeries V5.1 that do not require access to any of the new capabilities provided in version 5.1. The signature of this service program matches that contained in version 4.2.1.

QMQM/LIBMQM

This service program contains the single-threaded bindings for version 5.1 and above. See the following section for special considerations when writing threaded applications.

QMQM/LIBMQM_R

This service program contains the multi-threaded bindings for version 5.1 and above. See the following section for special considerations when writing threaded applications.

QMQM/LIBMQIC

This service program is for binding non-threaded client applications.

QMQM/LIBMQIC_R

This service program is for binding threaded client applications.

Use the CRTPGM command to create your programs. For example, the following command creates a single-threaded program that uses the ILE bound calls:

```
CRTPGM PGM(MYPROGRAM) BNDSRVPGM(QMQM/LIBMQM)
```

Threading considerations

The RPG compiler used for IBM i is part of the WebSphere Development Toolset and WebSphere Development Studio for IBM i and is known as the ILE RPG IV Compiler.

In general, RPG programs should not use the multi-threaded service programs. Exceptions are RPG programs created using the ILE RPG IV Compiler, and containing the THREAD(*SERIALIZE) keyword in the control specification. However, even though these programs are thread safe, careful consideration must be given to the overall application design, as THREAD(*SERIALIZE) forces serialization of RPG procedures at the module level, and this might have an adverse affect on overall performance.


Where RPG programs are used as data-conversion exits, they must be made thread-safe, and should be recompiled using the version 4.4 ILE RPG compiler or above, with THREAD(*SERIALIZE) specified in the control specification.

For further information about threading, see the *IBM i IBM MQ Development Studio: ILE RPG Reference*, and the *IBM i IBM MQ Development Studio: ILE RPG Programmer's Guide*.

Commitment control

The MQI syncpoint functions MQCMIT and MQBACK are available to ILE RPG programs running in normal mode; these calls allow the program to commit and back out changes to MQ resources.

The MQCMIT and MQBACK calls are not available to ILE RPG programs running in compatibility mode. For these programs, you should use the operation codes COMMIT and ROLBK.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Coding the bound calls

MQI ILE procedures are listed in [Table 574 on page 975](#).

Table 574. ILE RPG bound calls supported by each service program

Name of call	LIBMQM and LIBMQM_R	AMQZSTUB	AMQVSTUB	LIBMQIC and LIBMQIC_R
MQBACK	Y			Y
MQBEGIN	Y			Y
MQCMIT	Y			Y
MQCLOSE	Y	Y		Y
MQCONN	Y	Y		Y
MQCONNX	Y			Y
MQDISC	Y	Y		Y
MQGET	Y	Y		Y
MQINQ	Y	Y		Y
MQOPEN	Y	Y		Y
MQPUT	Y	Y		Y
MQPUT1	Y	Y		Y
MQSET	Y	Y		Y
MQXCNVC	Y		Y	Y

To use these procedures you need to:

1. Define the external procedures in your 'D' specifications. These are all available within the COPY file member CMQG containing the named constants.
2. Use the CALLP operation code to call the procedure along with its parameters.

For example the MQOPEN call requires the inclusion of the following code:

```

D*****
D**  MQOPEN Call -- Open Object (From COPY file CMQG)      **
D*****
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQOPEN          PR          EXTPROC('MQOPEN')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object descriptor
D OBJDSC          224A
D* Options that control the action of MQOPEN
D OPTS          10I 0 VALUE
D* Object handle
D HOBJ          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0
D*

```

To call the procedure, after initializing the various parameters, you need the following code:

```

...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
  C          CALLP      MQOPEN(HCONN : MQOD : OPTS : HOBJ :
  C          CMPCOD : REASON)

```

Here, the structure MQOD is defined using the COPY member CMQODG which breaks it down into its components.

Notational conventions

The latter topics in this section show how the:

- Calls should be invoked
- Parameters should be declared
- Various data types should be declared

In a number of cases, parameters are arrays or character strings with a size that is not fixed. For these, a lowercase "n" is used to represent a numeric constant. When the declaration for that parameter is coded, the "n" must be replaced by the numeric value required.

MQAIR - Authentication information record

The MQAIR structure represents the authentication information record.

Overview

Purpose: The MQAIR structure allows an application running as an IBM MQ client to specify information about an authenticator that is to be used for the client connection. The structure is an input parameter on the MQCONN call.

Character set and encoding: Data in MQAIR must be in the character set given by the *CodedCharSetId* queue-manager attribute and encoding of the local queue manager given by ENNAT.

- [“Fields” on page 976](#)
- [“Initial values” on page 978](#)
- [“RPG declaration” on page 978](#)

Fields

The MQAIR structure contains the following fields; the fields are described in **alphabetical order**:

AICN (10-digit signed integer)

This is either the host name or the network address of a host on which the LDAP server is running. This can be followed by an optional port number, enclosed in parentheses.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. If the value is not valid, the call fails with reason code RC2387.

The default port number is 389.

This is an input field. The length of this field is given by LNAICN. The initial value of this field is blank characters.

AITYP (10-digit signed integer)

This is the type of authentication information contained in the record.

The value must be:

AITLDP

Certificate revocation using LDAP server.

If the value is not valid, the call fails with reason code RC2386.

This is an input field. The initial value of this field is AITLDP.

AIPW (10-digit signed integer)

This is the password needed to access the LDAP CRL server.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. If the LDAP server does not require a password, or you omit the

LDAP user name, *AIPW* must be null or blank. If you omit the LDAP user name and *AIPW* is not null or blank, the call fails with reason code RC2390.

This is an input field. The length of this field is given by *LNLDPW*. The initial value of this field blank characters.

AILUL (10-digit signed integer)

This is the length in bytes of the LDAP user name addressed by the *AILUP* or *AILUO* field. The value must be in the range zero through *LNDISN*. If the value is not valid, the call fails with reason code RC2389.

If the LDAP server involved does not require a user name, set this field to zero.

This is an input field. The initial value of this field is 0.

AILUO (10-digit signed integer)

This is the offset in bytes of the LDAP user name from the start of the MQAIR structure.

The offset can be positive or negative. The field is ignored if *LDAPUserNameLength* is zero.

You can use either *LDAPUserNamePtr* or *LDAPUserNameOffset* to specify the LDAP user name, but not both; see the description of the *LDAPUserNamePtr* field for details.

This is an input field. The initial value of this field is 0.

AILUP (10-digit signed integer)

This is the LDAP user name.

It consists of the Distinguished Name of the user who is attempting to access the LDAP CRL server. If the value is shorter than the length specified by *AILUL*, terminate the value with a null character, or pad it with blanks to the length *AILUL*. The field is ignored if *AILUL* is zero.

You can supply the LDAP user name in one of two ways:

- By using the pointer field *AILUP*

In this case, the application can declare a string that is separate from the MQAIR structure, and set *AILUP* to the address of the string.

Consider using *AILUP* for programming languages that support the pointer data type in a fashion that is portable to different environments (for example, the C programming language).

- By using the offset field *AILUO*

In this case, the application must declare a compound structure containing the MQSCO structure followed by the array of MQAIR records followed by the LDAP user name strings, and set *AILUO* to the offset of the appropriate name string from the start of the MQAIR structure. Ensure that this value is correct, and has a value that can be accommodated within an MQLONG (the most restrictive programming language is COBOL, for which the valid range is -999 999 999 through +999 999 999).

Consider using *AILUO* for programming languages that do not support the pointer data type, or that implement the pointer data type in a fashion that might not be portable to different environments (for example, the COBOL programming language).

Whichever technique is chosen, use only one of *AILUP* and *AILUO* ; the call fails with reason code RC2388.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

AISID (10-digit signed integer)

The value must be:

AISIDV

Identifier for the authentication information record.

This is always an input field. The initial value of this field is AISIDV.

AIVER (10-digit signed integer)

The value must be:

AIVER1

Version-1 authentication information record.

The following constant specifies the version number of the current version:

AIRVERC

Current version of authentication information record.

This is always an input field. The initial value of this field is AIVER1.

Initial values

Table 575. Initial values of fields in MQAIR for MQAIR		
Field name	Name of constant	Value of constant
<i>AISID</i>	AISIDV	'AIR↵'
<i>AIVER</i>	AIVERC	1
<i>AITYP</i>	AITLDP	1
<i>AICN</i>	None	Null string or blanks
<i>AILUP</i>	None	Null pointer or null bytes
<i>AILUO</i>	None	0
<i>AILUL</i>	None	0
<i>AIPW</i>	None	Null string or blanks

Notes:

1. The symbol ↵ represents a single blank character.

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D* MQAIR Structure
D*
D* Structure identifier
D  AISID          1      4      INZ('AIR ')
D* Structure version number
D  AIVER          5      8I 0  INZ(1)
D* Type of authentication information
D  AITYP          9     12I 0  INZ(1)
D* Connection name of CRL LDAP server
D  AICN         13     276      INZ
D* Address of LDAP user name
D  AILUP        277     292*    INZ(*NULL)
D* Offset of LDAP user name from start of MQAIR structure
D  AILUO        293     296I 0  INZ(0)
D* Length of LDAP user name
D  AILUL        297     300I 0  INZ(0)
D* Password to access LDAP server
D  AIPW        301     332      INZ

```

MQBMHO - Buffer to message handle options

Structure defining the buffer to message handle options.

Overview

Purpose: The MQBMHO structure allows applications to specify options that control how message handles are produced from buffers. The structure is an input parameter on the MQBUFMH call.

Character set and encoding: Data in MQBMHO must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 979](#)
- [“Initial values” on page 980](#)
- [“RPG declaration” on page 980](#)

Fields

The MQBMHO structure contains the following fields; the fields are described in **alphabetical order**:

BMSID (10-digit signed integer)

Buffer to message handle structure - StrucId field.

This is the structure identifier. The value must be:

BMSIDV

Identifier for buffer to message handle structure.

This is always an input field. The initial value of this field is BMSIDV.

BMVER (10-digit signed integer)

Buffer to message handle structure - Version field.

This is the structure version number. The value must be:

BMVER1

Version number for buffer to message handle structure.

The following constant specifies the version number of the current version:

BMVERVC

Current version of buffer to message handle structure.

This is always an input field. The initial value of this field is BMVER1.

BMOPT (10-digit signed integer)

Buffer to message handle structure - Options field.

The value can be:

BMDLPR

Properties that are added to the message handle are deleted from the buffer. If the call fails no properties are deleted.

Default options: If you do not need the option described, use the following option:

BMNONE

No options specified.

This is always an input field. The initial value of this field is BMDLPR.

Initial values

Table 576. Initial values of fields in MQBMHO		
Field name	Name of constant	Value of constant
BMSID	BMSIDV	'BMHO'
BMVER	BMVER1	1
BMOPT	BMNONE	0

RPG declaration

```
D* MQBMHO Structure
D*
D*
D* Structure identifier
D  BMSID              1      4      INZ('BMHO')
D*
D* Structure version number
D  BMVER              5      8I 0  INZ(1)
D*
D* Options that control the action of MQBUFMH
D  BMOPT              9      12I 0 INZ(1)
```

MQBO - Begin options

The MQBO structure allows the application to specify options relating to the creation of a unit of work.

Overview

Purpose: The structure is an input/output parameter on the MQBEGIN call.

Character set and encoding: Data in MQBO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT.

- [“Fields” on page 980](#)
- [“Initial values” on page 981](#)
- [“RPG declaration” on page 981](#)

Fields

The MQBO structure contains the following fields; the fields are described in **alphabetical order**:

BOOPT (10-digit signed integer)

Options that control the action of MQBEGIN.

The value must be:

BONONE

No options specified.

This is always an input field. The initial value of this field is BONONE.

BOSID (4-byte character string)

Structure identifier.

The value must be:

BOSIDV

Identifier for begin-options structure.

This is always an input field. The initial value of this field is BOSIDV.

BOVER (10-digit signed integer)

Structure version number.

The value must be:

BOVER1

Version number for begin-options structure.

The following constant specifies the version number of the current version:

BOVERC

Current version of begin-options structure.

This is always an input field. The initial value of this field is BOVER1.

Initial values

Table 577. Initial values of fields in MQBO		
Field name	Name of constant	Value of constant
BOSID	BOSIDV	'B0 ' (B followed by a blank)
BOVER	BOVER1	1
BOOPT	BONONE	0

Notes:

1. The symbol – represents a single blank character.

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..  
D* MQBO Structure  
D*  
D* Structure identifier  
D BOSID 1 4 INZ('B0 ')  
D* Structure version number  
D BOVER 5 8I 0 INZ(1)  
D* Options that control the action of MQBEGIN  
D BOOPT 9 12I 0 INZ(0)
```

MQCBC - Callback context

Structure describing the callback routine.

Overview

Purpose

The MQCBC structure is used to specify context information that is passed to a callback function.

The structure is an input/output parameter on the call to a message consumer routine.

Version

The current version of MQCBC is CBCV2.

Character set and encoding

Data in MQCBC is in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure is in the character set and encoding of the client.

- [“Fields” on page 982](#)
- [“Initial values” on page 987](#)
- [“RPG declaration” on page 987](#)

Fields

The MQCBC structure contains the following fields; the fields are described in alphabetical order:

CBCBUFFLEN (10 digit signed integer)

The buffer can be larger than both the MaxMsgLength value defined for the consumer and the ReturnedLength value in the MQGMO.

Callback context structure - BufferLength field.

This is the length in bytes of the message buffer that has been passed to this function.

The actual message length is supplied in DataLength field.

The application can use the entire buffer for its own purposes for the duration of the callback function.

This is an input field to the message consumer function; it is not relevant to an exception handler function.

CBCCALLBA (10 digit signed integer)

Callback context structure - CallbackArea field.

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the CBDCALLBA field in the MQCBD structure, which is a parameter on the MQCB call used to define the callback function.

Changes to the *CBCCALLBA* are preserved across the invocations of the callback function for an *CBCHOBJ*. This field is not shared with callback functions for other handles.

This is an input/output field to the callback function. The initial value of this field is a null pointer or null bytes.

CBCCALLT (10 digit signed integer)

Callback Context structure - CallType field.

Field containing information about why this function has been called; the following are defined.

Message delivery call types: These call types contain information about a message. The *CBCLLEN* and *CBCBUFFLEN* parameters are valid for these call types.

CBCTMR

The message consumer function has been invoked with a message that has been destructively removed from the object handle.

If the value of *CBCCC* is CCWARN, the value of the *Reason* field is RC2079 or one of the codes indicating a data conversion problem.

CBCTMN

The message consumer function has been invoked with a message that has not yet been destructively removed from the object handle. The message can be destructively removed from the object handle using the *MsgToken*.

The message might not have been removed because:

- The MQGMO options requested a browse operation, GMBR*
- The message is larger than the available buffer and the MQGMO options do not specify *gmatm*

If the value of *CBCCC* is CCWARN, the value of the *Reason* field is RC2080 or one of the codes indicating a data conversion problem.

Callback control call types: These call types contain information about the control of the callback and do not contain details about a message. These call types are requested using CBDOPT in the MQCBD structure.

The *CBCLLEN* and *CBCBUFFLEN* parameters are not valid for these call types.

CBCTRC

The purpose of this call type is to allow the callback function to perform some initial setup.

The callback function is invoked immediately after the callback is registered, that is, upon return from an MQCB call using a value for the *Operation* field of CBREG.

This call type is used both for message consumers and event handlers.

If requested, this is the first invocation of the callback function.

The value of the *CBCREA* field is RCNONE.

CBCTSC

The purpose of this call type is to allow the callback function to perform some setup when it is started, for example, reinstating resources that were cleaned up when it was previously stopped.

The callback function is invoked when the connection is started using either CTLSR or CTLSW.

If a callback function is registered within another callback function, this call type is invoked when the callback returns.

This call type is used for message consumers only.

The value of the *CBCREA* field is RCNONE.

CBCTTC

The purpose of this call type is to allow the callback function to perform some cleanup when it is stopped for a while, for example, cleaning up additional resources that have been acquired during the consuming of messages.

The callback function is invoked when an MQCTL call is issued using a value for the *Operation* field of CTLSP.

This call type is used for message consumers only.

The value of the *CBCREA* field is set to indicate the reason for stopping.

CBCTDC

The purpose of this call type is to allow the callback function to perform final cleanup at the end of the consume process. The callback function is invoked when the:

- Callback function is deregistered using an MQCB call with BCUNR.
- Queue is closed, causing an implicit deregister. In this instance the callback function is passed HOUNUH as the object handle.
- MQDISC call completes - causing an implicit close and, therefore, a deregister. In this case the connection is not disconnected immediately, and any ongoing transaction is not yet committed.

If any of these actions are taken inside the callback function itself, the action is invoked once the callback returns.

This call type is used both for message consumers and event handlers.

If requested, this is the last invocation of the callback function.

The value of the *CBCREA* field is set to indicate the reason for stopping.

CBCTEC

Event handler function

The event handler function has been invoked without a message when:

- An MQCTL call is issued with a value for the *Operation* field of CTLSP, or
- The queue manager or connection stops or quiesces.

This call can be used to take appropriate action for all callback functions.

- **Message consumer function**

The message consumer function has been invoked without a message when an error (*CBCCC* = *CCFAIL*) has been detected that is specific to the object handle; for example *CBCREA* code = *RC2016* .

The value of the *CBCREA* field is set to indicate the reason for the call.

This is an input field. *CBCTMR* and *CMCTMN* are applicable only to message consumer functions.

CBCCC (10 digit signed integer)

Callback context structure - *CompCode* field.

This is the completion code. It indicates whether there were any problems consuming the message; it is one of the following:

CCOK

Successful completion

CCWARN

Warning (partial completion)

CCFAIL

Call failed

This is an input field. The initial value of this field is *CCOK*.

CBCCONNAREA (10 digit signed integer)

Callback context structure - *ConnectionArea* field.

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the *ConnectionArea* field in the *MQCTLO* structure, which is a parameter on the *MQCTL* call used to control the callback function.

Any changes made to this field by the callback functions are preserved across the invocations of the callback function. This area can be used to pass information that is to be shared by all callback functions. Unlike *CallbackArea*, this area is common across all callbacks for a connection handle.

This is an input and output field. The initial value of this field is a null pointer or null bytes.

CBCLLEN (10 digit signed integer)

This is the length in bytes of the application data in the message. If the value is zero, it means that the message contains no application data.

The *CBCLLEN* field contains the length of the message but not necessarily the length of the message data passed to the consumer. It could be that the message was truncated. Use the *GMRL* field in the *MQGMO* to determine how much data has been passed to the consumer.

If the reason code indicates the message has been truncated, you can use the *CBCLLEN* field to determine how large the actual message is. This allows you to determine the size of the buffer required to accommodate the message data, and then issue an *MQCB* call to update the *CBDMML* in the *MQCBD* with an appropriate value.

If the *GMCONV* option is specified, the converted message could be larger than the value returned for *DataLength*. In such cases, the application probably needs to issue an *MQCB* call to update the *CBDMML* in the *MQCBD* to be greater than the value returned by the queue manager for *DataLength*.

To avoid message truncation problems, specify *MaxMsgLength* as *CBDLFM*. This causes the queue manager to allocate a buffer for the full message length after data conversion. Be aware, however, that even if this option is specified, it is still possible that sufficient storage is not available to correctly process the request. Applications should always check the returned reason code. For example, if it is not possible to allocate sufficient storage to convert the message, the message is returned to the application unconverted.

This is an input field to the message consumer function; it is not relevant to an event handler function.

CBCFLG (10 digit signed integer)

Flags containing information about this consumer.

The following option is defined:

CBCFBE

This flag can be returned if a previous MQCLOSE call using the COQSC option failed with a reason code of RC2458.

This code indicated that the last read ahead message is being returned and that the buffer is now empty. If the application issues another MQCLOSE call using the COQSC option, it succeeds.

Note, that an application is not guaranteed to be given a message with this flag set, as there might still be messages in the read-ahead buffer that do not match the current selection criteria. In this instance, the consumer function is invoked with the reason code RC2019 .

If the read ahead buffer is empty, the consumer is invoked with the CBCFBE flag and the reason code RC2518.

This is an input field to the message consumer function; it is not relevant to an event handler function.

CBCHOBJ (10 digit signed integer)

Callback context structure - CBCHOBJ field.

For a call to a message consumer, this is the handle for the object relating to the message consumer.

For an event handler, this value is HONONE

The application can use this handle and the message token in the Get Message Options block to get the message if a message has not been removed from the queue.

This is always an input field. The initial value of this field is HOUNUH

CBCRCD (10 digit signed integer)

CBCRCD indicates how long the queue manager waits before trying to reconnect. The field can be modified by an event handler to change the delay or stop reconnection altogether.

Use the **CBCRCD** field only if the value of the **Reason** field in the Callback Context is RC2545.

On entry to the event handler the value of **CBCRCD** is the number of milliseconds the queue manager is going to wait before making a reconnection attempt. [Table 578 on page 985](#) lists the values that you can set to modify the behavior of the queue manager on return from the event handler.

Table 578. CBCRCD values	
Value	Description
-1	Make no more reconnection attempts. An error is returned to the application.
0	Try to reconnect immediately.
>0	Wait for this many milliseconds before trying the connection again.

CBCREA (10 digit signed integer)

Callback context structure - Reason field.

This is the reason code qualifying the CBCCC

This is an input field. The initial value of this field is RCNONE.

CBCSTATE (10 digit signed integer)

An indication of the state of the current consumer. This field is of most value to an application when a nonzero reason code is passed to the consumer function.

You can use this field to simplify application programming because you do not need to code behavior for each reason code.

This is an input field. The initial value of this field is CSNONE

State	Queue manager action	Value of constant
<i>CSNONE</i> This reason code represents a normal call with no additional reason information	None; this is the normal operation.	0
<i>CSSUST</i> These reason codes represent temporary conditions.	The callback routine is called to report the condition and then suspended. After a period the system might attempt the operation again, which can lead to the same condition being raised again.	1
<i>CSSUSU</i> These reason codes represent conditions where the callback needs to act to resolve the condition.	The consumer is suspended and the callback routine is called to report the condition. The callback routine should resolve the condition if possible and either RESUME or close down the connection.	2
<i>CSSUS</i> These reason codes represent failures that prevent further message callbacks.	The queue manager automatically suspends the callback function. If the callback function is resumed it is likely to receive the same reason code again.	3
<i>CSSTOP</i> These reason codes represent the end of message consumption.	Delivered to the exception handler and to callbacks that specified CBDTC. No further messages can be consumed.	4

CBCSID (10 digit signed integer)

Callback context structure - StrucId field.

This is the structure identifier; the value must be:

CBCSI

Identifier for callback context structure.

This is always an input field. The initial value of this field is CBCSI.

CBCVER (10 digit signed integer)

Callback context structure - Version field.

This is the structure version number; the value must be:

CBCV1

Version-1 callback context structure.

The following constant specifies the version number of the current version:

CBCCV

Current version of the callback context structure.

This is always an input field. The initial value of this field is CBCV1.

Initial values

Table 579. Initial values of fields in MQCBC		
Field name	Name of constant	Value of constant
<i>CBCSID</i>	CBCSI	'CBC~'
<i>CBCVER</i>	CBCV1	1
<i>CBCCALLT</i>	None	0
<i>CBCHOBJ</i>	HOUNUH	-1
<i>CBCCALLBA</i>	None	Null pointer or null bytes
<i>CBCCONNAREA</i>	None	Null pointer or null bytes
<i>CBCCC</i>	CCOK	0
<i>CBCREA</i>	RCNONE	0
<i>CBCSTATE</i>	CSNONE	0
<i>CBCLLEN</i>	None	0
<i>CBCBUFFLEN</i>	None	0
<i>CBCFLG</i>	None	0
<i>CBRCRD</i>	none	0

Note:

1. The symbol ~ represents a single blank character.

RPG declaration

```

D* MQCBC Structure
D*
D*
D* Structure identifier
D  CBCSID          1      4    INZ('CBC ')
D*
D* Structure version number
D  CBCVER          5      8I 0 INZ(1)
D*
D* Why Function was called
D  CBCCALLT        9     12I 0 INZ(0)
D*
D* Object Handle
D  CBCHOBJ        13     16I 0 INZ(-1)
D*
D* Callback data passed to the function
D  CBCCALLBA      17     32*  INZ(*NULL)
D*
D* MQCTL Data area passed to the function
D  CBCCONNAREA    33     48*  INZ(*NULL)
D*
D* Completion Code
D  CBCCC          49     52I 0 INZ(0)
D*
D* Reason Code
D  CBCREA         53     56I 0 INZ(0)
D*
D* Consumer State
D  CBCSTATE       57     60I 0 INZ(0)
D*
D* Message Data Length
D  CBCLLEN        61     64I 0 INZ(0)
D*
D* Buffer Length
D  CBCBUFFLEN     65     68I 0 INZ(0)
D*

```

```

** Flags containing information about
D* this consumer
D  CBCFLG              69      72I 0 INZ(0)
D* Ver:1 **
D* Number of milliseconds before reconnect attempt
D  CBCRCD              73      76I 0 INZ(0)
D* Ver:2 **
D*

```

MQCBD - Callback descriptor

Structure specifying the callback function.

Overview

Purpose: The MQCBD structure is used to specify a callback function and the options controlling its use by the queue manager.

The structure is an input parameter on the MQCB call.

Version: The current version of MQCBD is CBDV1.

Character set and encoding: Data in MQCBD must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and ENNAT. However, if the application is running as an IBM MQ MQI client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 988](#)
- [“Initial values” on page 992](#)
- [“RPG declaration” on page 992](#)

Fields

The MQCBD structure contains the following fields; the fields are described in **alphabetical order**:

CBDSCALLBA (10-digit signed integer)

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the field in the MQCBD structure, which is a parameter on the callback function declaration.

The value is used only on an *Operation* having a value CBREG, with no currently defined callback, it does not replace a previous definition.

This is an input and output field to the callback function. The initial value of this field is a null pointer or null bytes.

CBDSCALLBF (10-digit signed integer)

The callback function is invoked as a function call.

Use this field to specify a pointer to the callback function.

You *must* specify either *CallbackFunction* or *CallbackName*. If you specify both, the reason code RC2486 is returned.

If neither *CallbackName* nor *CallbackFunction* is not set, the call fails with the reason code RC2486.

This option is not supported in the following environments:

- CICS on z/OS
- Programming languages and compilers that do not support function-pointer references

In such situations, the call fails with the reason code RC2486.

This is an input field. The initial value of this field is a null pointer or null bytes.

CBDSCALLBN (10-digit signed integer)

The callback function is invoked as a dynamically linked program.

You *must* specify either *CallbackFunction* or *CallbackName*. If you specify both, the reason code RC2486 is returned.

If either *CallbackName* or *CallbackFunction* is not true, the call fails with the reason code RC2486.

The module is loaded when the first callback routine to use is registered, and unloaded when the last callback routine to use it deregisters.

Except where noted in the following text, the name is left-aligned within the field, with no embedded blanks; the name itself is padded with blanks to the length of the field. In the descriptions that follow, square brackets ([]) denote optional information:

IBMi

The callback name can be one of the following formats:

- Library "/" Program
- Library "/" ServiceProgram "("FunctionName")"

For example, MyLibrary/MyProgram(MyFunction).

The library name can be *LIBL. Both the library and program names are limited to a maximum of 10 characters.

UNIX systems

The callback name is the name of a dynamically loadable module or library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path:

```
[path]library(function)
```

If the path is not specified the system search path is used.

The name is limited to a maximum of 128 characters.

Windows

The callback name is the name of a dynamic-link library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path and drive:

```
[d:][path]library(function)
```

If the drive and path are not specified the system search path is used.

The name is limited to a maximum of 128 characters.

z/OS

The callback name is the name of a load module that is valid for specification on the EP parameter of the LINK or LOAD macro.

The name is limited to a maximum of 8 characters.

z/OS CICS

The callback name is the name of a load module that is valid for specification on the PROGRAM parameter of the EXEC CICS LINK command macro.

The name is limited to a maximum of 8 characters.

The program can be defined as remote using the REMOTESYTEM option of the installed PROGRAM definition or by the dynamic routing program.

The remote CICS region must be connected to IBM MQ if the program is to use IBM MQ API calls. Note, however, that the field in the MQCBC structure is not valid in a remote system.

If a failure occurs trying to load *CallbackName*, one of the following error codes is returned to the application:

- RC2495
- RC2496
- RC2497

A message is also written to the error log containing the name of the module for which the load was attempted, and the failing reason code from the operating system.

This is an input field. The initial value of this field is a null string or blanks.

CBDSCALLBT (10-digit signed integer)

This is the type of the callback function. The value must be one of:

CBTMC

Defines this callback as a message consumer function.

A message consumer callback function is called when a message, meeting the selection criteria specified, is available on an object handle and the connection is started.

CBTEH

Defines this callback as the asynchronous event routine; it is not driven to consume messages for a handle.

Hobj is not required on the MQCB call defining the event handler and is ignored if specified.

The event handler is called for conditions that affect the whole message consumer environment. The consumer function is invoked without a message when an event, for example, a queue manager or connection stopping, or quiescing, occurs. It is not called for conditions that are specific to a single message consumer, for example, RC2016.

Events are delivered to the application, regardless of whether the connection is started or stopped, except in the following environments:

- CICS on z/OS environment
- nonthreaded applications

If the caller does not pass one of these values, the call fails with a reason code of RC2483

This is always an input field. The initial value of this field is CBTMC.

CBDMML (10-digit signed integer)

This is the length in bytes of the longest message that can be read from the handle and given to the callback routine. If a message has a longer length, the callback routine receives *MaxMsgLength* bytes of the message, and reason code:

- RC2080 or
- RC2079 if you specified GMATM.

The actual message length is supplied in the [“CBCLEN \(10 digit signed integer\)”](#) on page 984 field of the MQCBC structure.

The following special value is defined:

CBDFFM

The buffer length is adjusted by the system to return messages without truncation.

If insufficient memory is available to allocate a buffer to receive the message, the system calls the callback function with an RC2071 reason code.

If, for example, you request data conversion, and there is insufficient memory available to convert the message data, the unconverted message is passed to the callback function.

This is an input field. The initial value of the *MaxMsgLength* field is CBDFM.

CBDOPT (10-digit signed integer)

Callback descriptor structure - Options field.

Any one, or all, of the following can be specified. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations). Combinations that are not valid are noted; any other combinations are valid.

CBDFQ

The MQCB call fails if the queue manager is in the quiescing state.

On z/OS, this option also forces the MQCB call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

Specify GMFIQ, in the MQGMO options passed on the MQCB call, to cause notification to message consumers when they are quiescing.

Control options: The following options control whether the callback function is called, without a message, when the state of the consumer changes:

CBDRRC

The callback function is invoked with call type CBCTRC

.

CBDSRC

The callback function is invoked with call type CBCTSC.

CBDTTC

The callback function is invoked with call type CBCTTC.

CBDDC

The callback function is invoked with call type CBCTDC.

See [“CBCCALLT \(10 digit signed integer\)” on page 982](#) for further details about these call types.

Default option: If you do not need any of the options described, use the following option:

CBDNO

Use this value to indicate that no other options have been specified; all options assume their default values.

CBDNO is defined to aid program documentation; it is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *Options* field is CBDNO.

CBDSID (10-digit signed integer)

Callback descriptor structure - StrucId field.

This is the structure identifier; the value must be:

CBDSI

Identifier for callback descriptor structure.

This is always an input field. The initial value of this field is CBDSI.

CBDVER (10-digit signed integer)

Callback descriptor structure - Version field.

This is the structure version number; the value must be:

CBDV1

Version-1 callback descriptor structure.

The following constant specifies the version number of the current version:

CBDV

Current version of callback descriptor structure.

This is always an input field. The initial value of this field is CBDV1.

Initial values

Table 580. Initial values of fields in MQCBD		
Field name	Name of constant	Value of constant
<i>StrucId</i>	CBDSI	'CBD~'
<i>Version</i>	CBDV1	1
<i>CallBackType</i>	CBTMC	1
<i>Options</i>	CBDNO	0
<i>CallbackArea</i>	None	Null bytes
<i>CallbackFunction</i>	None	Null bytes
<i>CallbackName</i>	None	Blanks
<i>MaxMsgLength</i>	CBDPM	-1

Note:

1. The symbol ~ represents a single blank character.

RPG declaration

```
D* MQCBD Structure
D*
D*
D* Structure identifier
D  CBDSID                1          4      INZ('CBD ')
D*
D* Structure version number
D  CBDVER                5          8I 0 INZ(1)
D*
D* Callback function type
D  CBDCALLBT            9          12I 0 INZ(1)
D*
** Options controlling message
D* consumption
D  CBDOPT               13          16I 0 INZ(0)
D*
D* User data passed to the function
D  CBDCALLBA           17          32*
D*
D* FP: Callback function pointer
D  CBDCALLBF           33          48*
D*
D* Callback name
D  CBDCALLBN           49          176      INZ('\0')
D*
D* Maximum message length
D  CBDMML             177          180I 0 INZ(-1)
```

MQCHARV - Variable Length String

Use the MQCHARV structure to describe a variable length string.

Overview

Character set and encoding: Data in the MQCHARV must be in the encoding of the local queue manager that is given by ENNAT and the character set of the VCHRC field within the structure. If the application

is running as an IBM MQ MQI client, the structure must be in the encoding of the client. Some character sets have a representation that depends on the encoding. If VCHRC is one of these character sets, the encoding used is the same encoding as that of the other fields in the MQCHARV. The character set identified by VSCCSID can be a double-byte character set (DBCS).

Usage: The MQCHARV structure addresses data that might be discontinuous with the structure containing it. To address this data, fields declared with the pointer data type can be used.

- [“Fields” on page 993](#)
- [“Initial values” on page 994](#)
- [“RPG declaration” on page 994](#)
- [“Redefinition of CSAPL” on page 994](#)

Fields

The MQCHARV structure contains the following fields; the fields are described in **alphabetical order**:

VCHRC (10-digit signed integer)

This is the character set identifier of the variable length string addressed by the VCHRP or VCHRO field.

The initial value of this field is CSAPL. This is defined by IBM MQ to indicate that it should be changed by the queue manager to the true character set identifier of the queue manager. This is in the same way as CSQM behaves. As a result, the value CSAPL is never associated with a variable length string. The initial value of this field can be changed by defining a different value for the constant CSAPL for your compile unit by the appropriate means for your application's programming language.

VCHRL (10-digit signed integer)

The length in bytes of the variable length string addressed by the VCHRP or VCHRO field.

The initial value of this field is 0. The value must be either greater than or equal to zero or the following special value which is recognized:

VSNTL

If VSNTL is not specified, VCHRL bytes are included as part of the string. If null characters are present they do not delimit the string.

If VSNTL is specified, the string is delimited by the first null encountered in the string. The null itself is not included as part of that string.

Note: The null character used to terminate a string if VSNTL is specified is a null from the code set specified by VCHRC.

For example, in UTF-16 (UCS-2 CCSIDs 1200 and 13488), this is the 2-byte Unicode encoding where a null is represented by a 16 bit number of all zeros. In UTF-16 it is common to find single bytes set to all zero which are part of characters (7-bit ASCII characters for example), but the strings will only be null terminated when two 'zero' bytes are found on an even byte boundary. It is possible to get two 'zero' bytes on an odd boundary when they are each part of valid characters. For example, x'01' x'00' x'00' x'30' represents two valid Unicode characters and does not null terminate the string.

VCHRO (10-digit signed integer)

The offset in bytes of the variable length string from the start of the MQCHARV, or the structure containing it.

When the MQCHARV structure is embedded within another structure, this value is the offset in bytes of the variable length string from the start of the structure that contains this MQCHARV structure. When the MQCHARV structure is not embedded within another structure, for example, if it is specified as a parameter on a function call, the offset is relative to the start of the MQCHARV structure.

The offset can be positive or negative. You can use either the VCHRP or VCHRO field to specify the variable length string, but not both.

The initial value of this field is 0.

VCHRP (pointer)

This is a pointer to the variable length string.

You can use either the VCHRP or VCHRO field to specify the variable length string, but not both.

The initial value of this field is a null pointer or null bytes.

VCHRS (10-digit signed integer)

The size in bytes of the buffer addressed by the VCHRP or VCHRO field.

When the MQCHARV structure is used as an output field on a function call, this field must be initialized with the length of the buffer provided. If the value of VCHRL is greater than VCHRS then only VCHRS bytes of data will be returned to the caller in the buffer.

The value must be greater than or equal to zero or the following special value which is recognized:

VSUSL

If VSUSL is specified, the length of the buffer is taken from the VCHRL field in the MQCHARV structure. This special value is not appropriate when the structure is used as an output field and a buffer is provided. This is the initial value of this field.

Initial values

Field name	Name of constant	Value of constant
VCHRP	None	Null pointer or null bytes.
VCHRO	None	0
VCHRS	VSUSL	-1
VCHRL	None	0
VCHRC	CSAPL	-3

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQCHARV Structure
D*
D* Address of variable length string
D VCHRP          1      16*
D* Offset of variable length string
D VCHRO          17      20I 0
D* Size of buffer
D VCHRS          21      24I 0
D* Length of variable length string
D VCHRL          25      28I 0
D* CCSID of variable length string
D VCHRC          29      32I 0
```

Redefinition of CSAPL

Unlike the programming languages supported on other platforms, RPG does not have a way of redefining a defined constant, so you must set each VCHRC specifically if you want to use a value other than CSAPL.

MQCIH - CICS bridge header

The MQCIH structure describes the information that can be present at the start of a message sent to the CICS bridge through IBM MQ for z/OS.

Overview

Format name: FMCICS.

Version: The current version of MQCIH is CIVER2. Fields that exist only in the more-recent version of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQCIH, with the initial value of the *CIVER* field set to CIVER2.

Character set and encoding: Special conditions apply to the character set and encoding used for the MQCIH structure and application message data:

- Applications that connect to the queue manager that owns the CICS bridge queue must provide an MQCIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQCIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQCIH structure that is in any of the supported character sets and encodings; conversion of the MQCIH is performed by the receiving message channel agent connected to the queue manager that owns the CICS bridge queue.

Note: There is one exception to this. If the queue manager that owns the CICS bridge queue is using CICS for distributed queuing, the MQCIH must be in the character set and encoding of the queue manager that owns the CICS bridge queue.

- The application message data following the MQCIH structure must be in the same character set and encoding as the MQCIH structure. The *CICSI* and *CIENC* fields in the MQCIH structure cannot be used to specify the character set and encoding of the application message data.

A data-conversion exit must be provided by the user to convert the application message data if the data is not one of the built-in formats supported by the queue manager.

Usage: If the values required by the application are the same as the initial values shown in [Table 582 on page 1004](#), and the bridge is running with AUTH=LOCAL or AUTH=IDENTIFY, the MQCIH structure can be omitted from the message. In all other cases, the structure must be present.

The bridge accepts either a version-1 or a version-2 MQCIH structure, but for 3270 transactions a version-2 structure must be used.

The application must ensure that fields documented as "request" fields have appropriate values in the message sent to the bridge; these fields are input to the bridge.

Fields documented as "response" fields are set by the CICS bridge in the reply message that the bridge sends to the application. Error information is returned in the *CIRET*, *CIFNC*, *CICC*, *CIREA*, and *CIAC* fields, but not all of them are set in all cases. [Table 581 on page 995](#) shows which fields are set for different values of *CIRET*.

Table 581. Contents of error information fields in MQCIH structure				
<i>CIRET</i>	<i>CIFNC</i>	<i>CICC</i>	<i>CIREA</i>	<i>CIAC</i>
CRC000	-	-	-	-
CRC003	-	-	FBC*	-
CRC002 CRC008	IBM MQ call name	IBM MQ <i>CMPCOD</i>	IBM MQ <i>REASON</i>	-
CRC001 CRC006 CRC007 CRC009	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	-
CRC004 CRC005	-	-	-	CICS ABCODE

- [“Fields” on page 996](#)
- [“Initial values” on page 1004](#)
- [“RPG declaration” on page 1005](#)

Fields

The MQCIH structure contains the following fields; the fields are described in **alphabetical order**:

CIAC (4-byte character string)

Abend code.

The value returned in this field is significant only if the *CIRET* field has the value CRC005 or CRC004. If it does, *CIAC* contains the CICS ABCODE value.

This is a response field. The length of this field is given by LNABNC. The initial value of this field is 4 blank characters.

This is an indicator specifying whether ADS descriptors should be sent on SEND and RECEIVE BMS requests. The following values are defined:

ADNONE

Do not send or receive ADS descriptor.

ADSEND

Send ADS descriptor.

ADRECV

Receive ADS descriptor.

ADMSGF

Use message format for the ADS descriptor.

This causes the ADS descriptor to be sent or received using the long form of the ADS descriptor. The long form has fields that are aligned on 4-byte boundaries.

The *CIADS* field should be set as follows:

- If ADS descriptors are *not* being used, set the field to ADNONE.
- If ADS descriptors *are* being used, and with the *same* CCSID in each environment, set the field to the sum of ADSEND and ADRECV.
- If ADS descriptors *are* being used, but with *different* CCSIDs in each environment, set the field to the sum of ADSEND, ADRECV, and ADMSGF.

This is a request field used only for 3270 transactions. The initial value of this field is ADNONE.

CIADS (10-digit signed integer)

Send/receive ADS descriptor.

This is an indicator specifying whether ADS descriptors should be sent on SEND and RECEIVE BMS requests. The following values are defined:

ADNONE

Do not send or receive ADS descriptor.

ADSEND

Send ADS descriptor.

ADRECV

Receive ADS descriptor.

ADMSGF

Use message format for the ADS descriptor.

This causes the ADS descriptor to be sent or received using the long form of the ADS descriptor. The long form has fields that are aligned on 4-byte boundaries.

The *CIADS* field should be set as follows:

- If ADS descriptors are *not* being used, set the field to ADNONE.
- If ADS descriptors *are* being used, and with the *same* CCSID in each environment, set the field to the sum of ADSEND and ADRECV.
- If ADS descriptors *are* being used, but with *different* CCSIDs in each environment, set the field to the sum of ADSEND, ADRECV, and ADMSGF.

This is a request field used only for 3270 transactions. The initial value of this field is ADNONE.

CIAI (4-byte character string)

AID key.

This is the initial value of the AID key when the transaction is started. It is a 1-byte value, left-aligned.

This is a request field used only for 3270 transactions. The length of this field is given by LNATID. The initial value of this field is 4 blanks.

CIAUT (8-byte character string)

Password or passticket.

This is a password or passticket. If user-identifier authentication is active for the CICS bridge, *CIAUT* is used with the user identifier in the MQMD identity context to authenticate the sender of the message.

This is a request field. The length of this field is given by LNAUTH. The initial value of this field is 8 blanks.

CICC (10-digit signed integer)

IBM MQ completion code or CICS EIBRESP.

The value returned in this field is dependent on *CIRET*; see [Table 581 on page 995](#).

This is a response field. The initial value of this field is CCOK.

CICNC (4-byte character string)

Abend transaction code.

This is the abend code to be used to terminate the transaction (normally a conversational transaction that is requesting more data). Otherwise this field is set to blanks.

This is a request field used only for 3270 transactions. The length of this field is given by LNCNCL. The initial value of this field is 4 blanks.

CICP (10-digit signed integer)

Cursor position.

This is the initial cursor position when the transaction is started. Later, for conversational transactions, the cursor position is in the RECEIVE vector.

This is a request field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *CIVER* is less than CIVER2.

CICSI (10-digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

CICT (10-digit signed integer)

Whether task can be conversational.

This is an indicator specifying whether the task should be allowed to issue requests for more information, or should abend. The value must be one of the following:

CTYES

Task is conversational.

CTNO

Task is not conversational.

This is a request field used only for 3270 transactions. The initial value of this field is CTNO.

CIENC (10-digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

CIEO (10-digit signed integer)

Offset of error in message.

This is the position of invalid data detected by the bridge exit. This field provides the offset from the start of the message to the location of the invalid data.

This is a response field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *CIVER* is less than *CIVER2*.

CIFAC (8-byte bit string)

Bridge facility token.

This is an 8-byte bridge facility token. The purpose of a bridge facility token is to allow multiple transactions in a pseudoconversation to use the same bridge facility (virtual 3270 terminal). In the first, or only, message in a pseudoconversation, a value of FCNONE should be set; this tells CICS to allocate a new bridge facility for this message. A bridge facility token is returned in response messages when a nonzero *CIFKT* is specified on the input message. Subsequent input messages can then use the same bridge facility token.

The following special value is defined:

FCNONE

No BVT token specified.

This is both a request and a response field used only for 3270 transactions. The length of this field is given by LNFAAC. The initial value of this field is FCNONE.

CIFKT (10-digit signed integer)

Bridge facility release time.

This is the length of time in seconds that the bridge facility will be kept after the user transaction has ended. For nonconversational transactions, the value should be zero.

This is a request field used only for 3270 transactions. The initial value of this field is 0.

CIFL (4-byte character string)

Terminal emulated attributes.

This is the name of an installed terminal that is to be used as a model for the bridge facility. A value of blanks means that *CIFL* is taken from the bridge transaction profile definition, or a default value is used.

This is a request field used only for 3270 transactions. The length of this field is given by LNFAAC. The initial value of this field is 4 blanks.

CIFLG (10-digit signed integer)

Flags.

The value must be:

CIFNON

No flags.

This is a request field. The initial value of this field is CIFNON.

CIFMT (8-byte character string)

IBM MQ format name of data that follows MQCIH.

This specifies the IBM MQ format name of the data that follows the MQCIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

This format name is also used for the reply message, if the *CIRFM* field has the value FMNONE.

- For DPL requests, *CIFMT* must be the format name of the COMMAREA.
- For 3270 requests, *CIFMT* must be CSQCBDCI, and *CIRFM* must be CSQCBDCO.

The data-conversion exits for these formats must be installed on the queue manager where they are to run.

If the request message results in the generation of an error reply message, the error reply message has a format name of FMSTR.

This is a request field. The length of this field is given by LNFMT. The initial value of this field is FMNONE.

CIFNC (4-byte character string)

IBM MQ call name or CICS EIBFN function.

The value returned in this field is dependent on *CIRET*; see [Table 581 on page 995](#). The following values are possible when *CIFNC* contains an IBM MQ call name:

CFCONN

MQCONN call.

CFGET

MQGET call.

CFINQ

MQINQ call.

CFOPEN

MQOPEN call.

CFPUT

MQPUT call.

CFPUT1

MQPUT1 call.

CFNONE

No call.

This is a response field. The length of this field is given by LNFUNC. The initial value of this field is CFNONE.

CIGWI (10-digit signed integer)

Wait interval for MQGET call issued by bridge task.

This field is applicable only when *CIUOW* has the value CUFRST. It allows the sending application to specify the approximate time in milliseconds that the MQGET calls issued by the bridge should wait for second and subsequent request messages for the unit of work started by this message. This overrides the default wait interval used by the bridge. The following special values may be used:

WIDFLT

Default wait interval.

This causes the CICS bridge to wait for the period specified when the bridge was started.

WIULIM

Unlimited wait interval.

This is a request field. The initial value of this field is WIDFLT.

CIHI (10-digit signed integer)

Reserved.

This is a reserved field. The value must be 0. This field is not present if *CIVER* is less than *CIVER2*.

CILEN (10-digit signed integer)

Length of MQCIH structure.

The value must be one of the following:

CILEN1

Length of version-1 CICS information header structure.

CILEN2

Length of version-2 CICS information header structure.

The following constant specifies the length of the current version:

CILENC

Length of current version of CICS information header structure.

This is a request field. The initial value of this field is *CILEN2*.

CILT (10-digit signed integer)

Link type.

This indicates the type of object that the bridge should try to link. The value must be one of the following:

LTPROG

DPL program.

LTTRAN

3270 transaction.

This is a request field. The initial value of this field is *LTPROG*.

CINTI (4-byte character string)

Next transaction to attach.

This is the name of the next transaction returned by the user transaction (typically by EXEC CICS RETURN TRANSID). If there is no next transaction, this field is set to blanks.

This is a response field used only for 3270 transactions. The length of this field is given by *LNTRID*. The initial value of this field is 4 blanks.

CIODL (10-digit signed integer)

Output COMMAREA data length.

This is the length of the user data to be returned to the client in a reply message. This length includes the 8-byte program name. The length of the COMMAREA passed to the linked program is the maximum of this field and the length of the user data in the request message, minus 8.

Note: The length of the user data in a message is the length of the message *excluding* the MQCIH structure.

If the length of the user data in the request message is smaller than *CIODL*, the *DATALength* option of the LINK command is used; this allows the LINK to be function-shipped efficiently to another CICS region.

The following special value can be used:

OLINPT

Output length is same as input length.

This value might be needed even if no reply is requested, in order to ensure that the COMMAREA passed to the linked program is of sufficient size.

This is a request field used only for DPL programs. The initial value of this field OLINPT.

CIREA (10-digit signed integer)

IBM MQ reason or feedback code, or CICS EIBRESP2.

The value returned in this field is dependent on *CIRET* ; see [Table 581 on page 995](#).

This is a response field. The initial value of this field is RCNONE.

CIRET (10-digit signed integer)

Return code from bridge.

This is the return code from the CICS bridge describing the outcome of the processing performed by the bridge. The *CIFNC*, *CICC*, *CIREA*, and *CIAC* fields may contain additional information (see [Table 581 on page 995](#)). The value is one of the following:

CRC000

(0, X'000') No error.

CRC001

(1, X'001') EXEC CICS statement detected an error.

CRC002

(2, X'002') IBM MQ call detected an error.

CRC003

(3, X'003') CICS bridge detected an error.

CRC004

(4, X'004') CICS bridge ended abnormally.

CRC005

(5, X'005') Application ended abnormally.

CRC006

(6, X'006') Security error occurred.

CRC007

(7, X'007') Program not available.

CRC008

(8, X'008') Second or later message within current unit of work not received within specified time.

CRC009

(9, X'009') Transaction not available.

This is a response field. The initial value of this field is CRC000.

CIRFM (8-byte character string)

IBM MQ format name of reply message.

This is the IBM MQ format name of the reply message that will be sent in response to the current message. The rules for coding this are the same as those for the *MDFMT* field in MQMD.

This is a request field used only for DPL programs. The length of this field is given by LNFMT. The initial value of this field is FMNONE.

CIRSI (4-byte character string)

Reserved.

This is a reserved field. The value must be 4 blanks. The length of this field is given by LNRSID.

CIRS1 (8-byte character string)

Reserved.

This is a reserved field. The value must be 8 blanks.

CIRS2 (8-byte character string)

Reserved.

This is a reserved field. The value must be 8 blanks.

CIRS3 (8-byte character string)

Reserved.

This is a reserved field. The value must be 8 blanks.

CIRS4 (10-digit signed integer)

Reserved.

This is a reserved field. The value must be 0. This field is not present if *CIVER* is less than *CIVER2*.

CIRTI (4-byte character string)

Reserved.

This is a reserved field. The value must be 4 blanks. The length of this field is given by *LNTRID*.

CISC (4-byte character string)

Transaction start code.

This is an indicator specifying whether the bridge emulates a terminal transaction or a START transaction. The value must be one of the following:

SCSTRT

Start.

SCDATA

Start data.

SCTERM

Terminate input.

SCNONE

None.

In the response from the bridge, this field is set to the start code appropriate to the next transaction ID contained in the *CINTI* field. The following start codes are possible in the response:

- SCSTRT
- SCDATA
- SCTERM

For CICS Transaction Server Version 1.2, this field is a request field only; its value in the response is undefined.

For CICS Transaction Server Version 1.3 and subsequent releases, this is both a request and a response field.

This field is used only for 3270 transactions. The length of this field is given by *LNSTCO*. The initial value of this field is *SCNONE*.

CISID (4-byte character string)

Structure identifier.

The value must be:

CISIDV

Identifier for CICS information header structure.

This is a request field. The initial value of this field is CISIDV.

CITES (10-digit signed integer)

Status at end of task.

This field shows the status of the user transaction at end of task. One of the following values is returned:

TENOSY

Not synchronized.

The user transaction has not yet completed and has not syncpointed. The *MDMT* field in MQMD is MTRQST in this case.

TECMIT

Commit unit of work.

The user transaction has not yet completed, but has syncpointed the first unit of work. The *MDMT* field in MQMD is MTDGRM in this case.

TEBACK

Back out unit of work.

The user transaction has not yet completed. The current unit of work will be backed out. The *MDMT* field in MQMD is MTDGRM in this case.

TEENDT

End task.

The user transaction has ended (or abended). The *MDMT* field in MQMD is MTRPLY in this case.

This is a response field used only for 3270 transactions. The initial value of this field is TENOSY.

CITI (4-byte character string)

Transaction to attach.

If *CILT* has the value LTTRAN, *CITI* is the transaction identifier of the user transaction to be run; a nonblank value must be specified in this case.

If *CILT* has the value LTPROG, *CITI* is the transaction code under which all programs within the unit of work are to be run. If the value specified is blank, the CICS DPL bridge default transaction code (CKBP) is used. If the value is nonblank, it must have been defined to CICS as a local TRANSACTION with an initial program of CSQCBP00. This field is applicable only when *CIUOW* has the value CUFRST or CUONLY.

This is a request field. The length of this field is given by LNTRID. The initial value of this field is 4 blanks.

CIUOW (10-digit signed integer)

Unit-of-work control.

This controls the unit-of-work processing performed by the CICS bridge. You can request the bridge to run a single transaction, or one or more programs within a unit of work. The field indicates whether the CICS bridge should start a unit of work, perform the requested function within the current unit of work, or end the unit of work by committing it or backing it out. Various combinations are supported, to optimize the data transmission flows.

The value must be one of the following:

CUONLY

Start unit of work, perform function, then commit the unit of work (DPL and 3270).

CUCONT

Additional data for the current unit of work (3270 only).

CUFRST

Start unit of work and perform function (DPL only).

CUMIDL

Perform function within current unit of work (DPL only).

CULAST

Perform function, then commit the unit of work (DPL only).

CUCMIT

Commit the unit of work (DPL only).

CUBACK

Back out the unit of work (DPL only).

This is a request field. The initial value of this field is CUONLY.

CIVER (10-digit signed integer)

Structure version number.

The value must be one of the following:

CIVER1

Version-1 CICS information header structure.

CIVER2

Version-2 CICS information header structure.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

CIVERC

Current version of CICS information header structure.

This is a request field. The initial value of this field is CIVER2.

Initial values

<i>Table 582. Initial values of fields in MQCIH</i>		
Field name	Name of constant	Value of constant
<i>CISID</i>	CISIDV	'CIH~'
<i>CIVER</i>	CIVER2	2
<i>CILEN</i>	CILEN2	180
<i>CIENC</i>	None	0
<i>CICSI</i>	None	0
<i>CIFMT</i>	FMNONE	Blanks
<i>CIFLG</i>	CIFNON	0
<i>CIRET</i>	CRC000	0
<i>CICC</i>	CCOK	0
<i>CIREA</i>	RCNONE	0
<i>CIUOW</i>	CUONLY	273
<i>CIGWI</i>	WIDFLT	-2
<i>CILT</i>	LTPROG	1
<i>CIODL</i>	OLINPT	-1
<i>CIFKT</i>	None	0
<i>CIADS</i>	ADNONE	0

Table 582. Initial values of fields in MQCIH (continued)

Field name	Name of constant	Value of constant
<i>CICT</i>	CTNO	0
<i>CITES</i>	TENOSY	0
<i>CIFAC</i>	FCNONE	Nulls
<i>CIFNC</i>	CFNONE	Blanks
<i>CIAC</i>	None	Blanks
<i>CIAUT</i>	None	Blanks
<i>CIRS1</i>	None	Blanks
<i>CIRFM</i>	FMNONE	Blanks
<i>CIRSI</i>	None	Blanks
<i>CIRTI</i>	None	Blanks
<i>CITI</i>	None	Blanks
<i>CIFL</i>	None	Blanks
<i>CIAI</i>	None	Blanks
<i>CISC</i>	SCNONE	Blanks
<i>CICNC</i>	None	Blanks
<i>CINTI</i>	None	Blanks
<i>CIRS2</i>	None	Blanks
<i>CIRS3</i>	None	Blanks
<i>CICP</i>	None	0
<i>CIEO</i>	None	0
<i>CIII</i>	None	0
<i>CIRS4</i>	None	0

Notes:

1. The symbol ~ represents a single blank character.

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D* MQCIH Structure
D*
D* Structure identifier
D  CISID          1      4      INZ('CIH ')
D* Structure version number
D  CIVER          5      8I 0 INZ(2)
D* Length of MQCIH structure
D  CILEN          9      12I 0 INZ(180)
D* Reserved
D  CIENC         13      16I 0 INZ(0)
D* Reserved
D  CICSI         17      20I 0 INZ(0)
D* MQ format name of data that followsMQCIH
D  CIFMT         21      28      INZ(' ')
D* Flags
D  CIFLG         29      32I 0 INZ(0)
D* Return code from bridge

```

```

D CIRET 33 36I 0 INZ(0)
D* MQ completion code or CICSEIBRESP
D CICC 37 40I 0 INZ(0)
D* MQ reason or feedback code, or CICSEIBRESP2
D CIREA 41 44I 0 INZ(0)
D* Unit-of-work control
D CIUOW 45 48I 0 INZ(273)
D* Wait interval for MQGET call issuedby bridge task
D CIGWI 49 52I 0 INZ(-2)
D* Link type
D CILT 53 56I 0 INZ(1)
D* Output COMMAREA data length
D CIODL 57 60I 0 INZ(-1)
D* Bridge facility release time
D CIFKT 61 64I 0 INZ(0)
D* Send/receive ADS descriptor
D CIADS 65 68I 0 INZ(0)
D* Whether task can beconversational
D CICT 69 72I 0 INZ(0)
D* Status at end of task
D CITES 73 76I 0 INZ(0)
D* Bridge facility token
D CIFAC 77 84 INZ(X'00000000000000-
D 00')
D* MQ call name or CICS EIBFNfunction
D CIFNC 85 88 INZ(' ')
D* Abend code
D CIAC 89 92 INZ
D* Password or passticket
D CIAUT 93 100 INZ
D* Reserved
D CIRS1 101 108 INZ
D* MQ format name of reply message
D CIRFM 109 116 INZ(' ')
D* Remote CICS system id to use
D CIRSI 117 120 INZ
D* CICS RTRANSID to use
D CIRTI 121 124 INZ
D* Transaction to attach
D CITI 125 128 INZ
D* Terminal emulated attributes
D CIFL 129 132 INZ
D* AID key
D CIAI 133 136 INZ
D* Transaction start code
D CISC 137 140 INZ(' ')
D* Abend transaction code
D CICNC 141 144 INZ
D* Next transaction to attach
D CINTI 145 148 INZ
D* Reserved
D CIRS2 149 156 INZ
D* Reserved
D CIRS3 157 164 INZ
D* Cursor position
D CICP 165 168I 0 INZ(0)
D* Offset of error in message
D CIEO 169 172I 0 INZ(0)
D* Reserved
D CIII 173 176I 0 INZ(0)
D* Reserved
D CIRS4 177 180I 0 INZ(0)
D*

```

MQCMHO - Create message handle options

The **MQCMHO** structure allows applications to specify options that control how message handles are created.

Overview

Purpose

The structure is an input parameter on the **MQCRTMH** call.

Character set and encoding

Data in **MQCMHO** must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1007](#)
- [“Initial values” on page 1008](#)
- [“RPG declaration” on page 1008](#)

Fields

The MQCMHO structure contains the following fields; the fields are described in alphabetical order:

CMOPT (10 digit signed integer)

One of the following options can be specified:

CMVAL

When **MQSETMP** is called to set a property in this message handle, the property name is validated to ensure that it:

- contains no invalid characters.
- does not begin "JMS" or "usr.JMS" except for the following:
 - JMSCorrelationID
 - JMSReplyTo
 - JMSType
 - JMSXGroupID
 - JMSXGroupSeq

These names are reserved for JMS properties.

- is not one of the following keywords, in any mixture of upper or lowercase:
 - "AND"
 - "BETWEEN"
 - "ESCAPE"
 - "FALSE"
 - "IN"
 - "IS"
 - "LIKE"
 - "NOT"
 - "NULL"
 - "OR"
 - "TRUE"
- does not begin "Body." or "Root." (except for "Root.MQMD").

If the property is MQ-defined ("mq.*") and the name is recognized, the property descriptor fields are set to the correct values for the property. If the property is not recognized, the *Support* field of the property descriptor is set to **PDSUPO** (for more information, see [PDSUP](#)).

CMDEFV

This specifies that the default level of validation of property names occurs.

The default level of validation is equivalent to that specified by **CMVAL**.

In a future release an administrative option might be defined which will change the level of validation that will occur when **CMDEFV** is defined.

This is the default value.

CMNOVA

No validation on the property name occurs. See the description of **CMVAL**.

Default option: If none of the options previously described in this section is required, the following option can be used:

CMNONE

All options assume their default values. Use this value to indicate that no other options have been specified. **CMNONE** aids program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is always an input field. The initial value of this field is **CMDEFV**.

CMSID (10 digit signed integer)

This is the structure identifier; the value must be:

CMSIDV

Identifier for create message handle options structure.

This is always an input field. The initial value of this field is **CMSIDV**.

CMVER (10 digit signed integer)

This is the structure version number; the value must be:

CMVER1

Version-1 create message handle options structure.

The following constant specifies the version number of the current version:

CMVERC

Current version of create message handle options structure.

This is always an input field. The initial value of this field is **CMVER1**.

Initial values

Table 583. Initial values of fields in MQCMHO		
Field name	Name of constant	Value of constant
CMSID	CMSIDV	'CMHO'
CMVER	CMVER1	1
CMOPT	CMDEFV	0

RPG declaration

```
D* MQCMHO Structure
D*
D*
D* Structure identifier
D  CMSID          1      4    INZ('CMHO')
D*
D* Structure version number
D  CMVER          5      8I 0 INZ(1)
D*
D* Options that control the action of MQCRTMH
D  CMOPT          9     12I 0 INZ(0)
```

MQCNO - Connect options

The MQCNO structure allows the application to specify options relating to the connection to the local queue manager.

Overview

Purpose: The structure is an input/output parameter on the MQCONN call.

Version: The current version of MQCNO is CNVER5. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQCNO that is supported by the environment, but with the initial value of the *CNVER* field set to CNVER1. To use fields that are not present in the version-1 structure, the application must set the *CNVER* field to the version number of the version required.

Character set and encoding: Data in MQCNO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT.

- [“Fields” on page 1009](#)
- [“Initial values” on page 1013](#)
- [“RPG declaration” on page 1014](#)

Fields

The MQCNO structure contains the following fields; the fields are described in **alphabetical order**:

CNCCO (10-digit signed integer)

This is the offset in bytes of an MQCD channel definition structure from the start of the MQCNO structure.

CNCCP (pointer)

This is a pointer to an MQCD channel definition structure.

CNCONID (24-byte character string)

Unique connection identifier. This field allows the queue manager to reliably identify an application process by assigning it a unique identifier when it first connects to the queue manager.

Applications use the connection identifier for correlation purposes when making PUT and GET calls. All connections are assigned an identifier by the queue manager, no matter how the connection was established.

It is possible to use the connection identifier to force the end of a long running unit of work. To do this, specifying the connection identifier using the PCF command 'Stop Connection', or the MQSC command STOP CONN. For more information about using these commands, see the related links.

The initial value of the field is 24 null bytes.

CNCT (128-byte bit string)

This is a tag that the queue manager associates with the resources that are affected by the application during this connection.

Queue-manager connection tag.

Each application or application instance must use a different value for the tag, so that the queue manager can correctly serialize access to the affected resources. See the descriptions of the CN*CT* options for further details. The tag ceases to be valid when the application terminates, or issues the MQDISC call.

Use the following special value if no tag is required:

CTNONE

No connection tag specified.

The value is binary zero for the length of the field.

This is an input field. The length of this field is given by LNCTAG. The initial value of this field is CTNONE. This field is ignored if CNVER is less than CNVER3.

Use the field ConnTag when connecting to a z/OS queue manager.

CNOPT (10 digit signed integer)

Options that control the action of MQCONN.

Binding options

The binding options control the type of IBM MQ binding that is used; specify only one of these options:

CNSBND

Standard binding.

The standard binding option causes the application and the local queue manager agent to run in separate units of execution, typically in separate processes. The arrangement maintains the integrity of the queue manager; that is, it protects the queue manager from errant programs.

Use CNSBND in situations where the application might not have been fully tested, or might be unreliable or untrustworthy. CNSBND is the default.

CNSBND is defined to aid program documentation. Do not use this option with any other option controlling the type of binding used; but because its value is zero, such use cannot be detected.

This option is supported in all environments.

CNFBND

Fast path binding.

The fast path binding option causes the application and the local queue manager agent to be part of the same unit of execution. Fast path is in contrast to the standard binding, where the application and the local-queue manager agent run in separate units of execution.

CNFBND is ignored if the queue manager does not support this type of binding; processing continues as though the option had not been specified.

CNFBND can be of advantage in situations where multiple processes consume more resources than the overall resource used by the application. An application that uses the fast path binding is known as a *trusted application*.

Consider the following important points when deciding whether to use the fast path binding:

- **Using the CNFBND option does not prevent an application altering or corrupting messages and other data areas belonging to the queue manager. Use this option only in situations where you have fully evaluated these issues.**
- The application must not use asynchronous signals or timer interrupts (such as sigkill) with CNFBND. There are also restrictions on the use of shared memory segments.
- The application must not have more than one thread connected to the queue manager at any one time.
- The application must use the MQDISC call to disconnect from the queue manager.
- The application must finish before ending the queue manager with the endmqm command.

The following points apply to the use of CNFBND in the environments indicated:

- On IBM i, the job must run under user profile QMQM that belongs to the QMQMADM group. Also, the program must not terminate abnormally, otherwise unpredictable results might occur.

For more information about the implications of using trusted applications, see [Connecting to a queue manager using the MQCONN call](#) and [Restrictions for trusted applications](#).

CNSHBD

Shared Bindings.

The shared bindings option causes the application and the local queue manager agent to run in separate units of execution, typically in separate processes. The arrangement maintains the integrity of the queue manager; that is, it protects the queue manager from errant programs. However some resources are shared between the application and the local queue manager agent. CNSHBD is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

CNIBND

Isolated Bindings.

The isolated bindings option causes the application and the local queue manager agent to run in separate units of execution, typically in separate processes. The arrangement maintains the integrity of the queue manager; that is, it protects the queue manager from errant programs. The application process and the local queue manager agent are isolated from each other in that they do not share resources. CNIBND is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

Handle-sharing options

The following options control the sharing of handles between different threads (units of parallel processing) within the same process. Only one of these options can be specified.

CNHSN

No handle sharing between threads.

The no handle sharing between threads option indicates that connection and object handles can be used only by the thread that caused the handle to be allocated; that is, the thread that issued the MQCONN, MQCONNX, or MQOPEN call. The handles cannot be used by other threads belonging to the same process.

CNHSB

Serial handle sharing between threads, with call blocking.

The serial handle sharing between threads, with call blocking, option indicates that connection and object handles allocated by one thread of a process can be used by other threads belonging to the same process. However, only one thread at a time can use any particular handle, that is, only serial use of a handle is permitted. If a thread tries to use a handle that is already in use by another thread, the call blocks (waits) until the handle becomes available.

CNHSNB

Serial handle sharing between threads, without call blocking.

The serial handle sharing between threads, *without* call blocking, option is the same as the "*with* blocking" option, except that, if the handle is in use by another thread, the call completes immediately with CCFAIL and RC2219 instead of blocking until the handle becomes available.

A thread can have zero or one nonshared handles, plus zero or more shared handles:

- Each MQCONN or MQCONNX call that specifies CNHSN returns a new nonshared handle on the first call, and the same nonshared handle on subsequent calls (assuming no intervening MQDISC call). The reason code is RC2002 for the second and later calls.
- Each MQCONNX call that specifies CNHSB or CNHSNB returns a new shared handle on each call.

Object handles inherit the same sharing properties as the connection handle specified on the MQOPEN call that created the object handle. Also, units of work inherit the same sharing properties as the connection handle used to start the unit of work; if the unit of work is started

in one thread using a shared handle, the unit of work can be updated in another thread using the same handle.

If you do not specify a handle-sharing option, the default is determined by the environment:

- In the Microsoft Transaction Server (MTS) environment, the default is the same as CNHSB.
- In other environments, the default is the same as CNHSN.

Reconnection options

Reconnection options determine if a connection is reconnectable. Only client connections are reconnectable.

CNRCDF

The reconnection option is resolved to its default value. If no default is set, the value of this option resolves to **DISABLED**. The value of the option is passed to the server, and can be queried by **PCF** and **MQSC**.

CNRC

The application can be reconnected to any queue manager consistent with the value of the MQCONN **QMNAME** parameter. Use the CNRC option only if there is no affinity between the client application and the queue manager with which it initially established a connection. The value of the option is passed to the server, and can be queried by **PCF** and **MQSC**.

CNRCD

The application cannot be reconnected. The value of the option is *not* passed to the server.

CNRCQM

The application can only be reconnected to the queue manager with which it originally connected. Use this value if a client can be reconnected, but there is an affinity between the client application, and the queue manager with which it originally established a connection. Choose this value if you want a client to automatically reconnect to the standby instance of a highly available queue manager. The value of the option is passed to the server, and can be queried by **PCF** and **MQSC**.

Use the options CNRC, CNRCD, and CNRCQM only for client connections. If the options are used for a binding connection, MQCONN fails with completion code, MQCC_FAILED and reason code, MQRC_OPTIONS_ERROR.

Default option: If none of the options described is required, the following option can be used:

CNNONE

No options are specified.

CNNONE is defined to aid program documentation. It is not intended that this option is used with any other CN* option, but because its value is zero, such use cannot be detected.

CNSCO (10-digit signed integer)

This is the offset in bytes of an MQSCO structure from the start of the MQCNO structure.

This field is ignored if *CNVER* is less than CNVER4.

CNSCP (pointer)

This is the address of an MQSCO structure.

This field is ignored if *CNVER* is less than CNVER4.

CNSECPO (10-digit signed integer)

Security parameters offset. The offset of the MQCSP structure used for specifying a user ID and password.

The value may be positive or negative. The initial value of this field is 0.

This field is ignored if *CNVER* is less than CNVER5.

CNSECPP (pointer)

Security parameters pointer. Address of the MQCSP structure used for specifying a user ID and a password.

The initial value of this field is a null pointer or null bytes.

This field is ignored if *CNVER* is less than *CNVER5*.

CNSID (4-byte character string)

The structure identifier for the MQCNO structure.

The value must be:

CNSIDV

Identifier for connect-options structure.

This is always an input field. The initial value of this field is *CNSIDV*.

CNVER (10-digit signed integer)

The structure version number for the MQCNO structure.

The value must be:

CNVER5

Version-5 connect-options structure.

This version is supported in all environments.

The following constant specifies the version number of the current version:

CNVERC

Current version of connect-options structure.

This is always an input field. The initial value of this field is *CNVER5*.

Initial values

<i>Table 584. Initial values of fields in MQCNO</i>		
Field name	Name of constant	Value of constant
<i>CNSID</i>	<i>CNSIDV</i>	'CNO–
<i>CNVER</i>	<i>CNVER5</i>	1
<i>CNOPT</i>	<i>CNNONE</i>	0
<i>CNCCO</i>	None	0
<i>CNCCP</i>	None	Null pointer or null bytes
<i>CNCT</i>	<i>CTNONE</i>	Nulls
<i>CNSCP</i>	None	Null pointer or null bytes
<i>CNSCO</i>	None	0
<i>CNCONID</i>	None	Nulls
<i>CNSECPO</i>	None	0
<i>CNSECPP</i>	None	Null pointer or null bytes

Notes:

1. The symbol – represents a single blank character.

[illegible]

Summary of the MQCSP structure for IBM MQ for IBM i.

Purpose: The MQCSP structure enables the authorization service to authenticate a user ID and password. You specify the MQCSP connection security parameters structure on an MQCONN call.

- “Fields” on page 1014
- “Initial values” on page 1016
- “RPG declaration” on page 1016

The MQCSP structure contains the following fields; the fields are described in **alphabetical order**:

This is the type of authentication to perform.

CSAN

1014 IBM MQ Developing Applications Reference

CSAUIAP

Authenticate user ID and password fields.

This is an input field. The initial value of this field is CSAN.

CSCPPL (10-digit signed integer)

This is the length of the password to be used in authentication.

The maximum length of the password is not dependent on the platform. If the length of the password is greater than that allowed, the authentication request fails with an RC2035.

This is an input field. The initial value of this field is 0.

CSCPPO (10-digit signed integer)

This is the offset in bytes of the password to be used in authentication.

The offset can be positive or negative.

This is an input field. The initial value of this field is 0.

CSCPPP (pointer)

This is the address of the password to be used in authentication.

This is an input field. The initial value of this field is the null pointer.

CSCSPUIL (10-digit signed integer)

This is the length of the user ID to be used in authentication.

The maximum length of the user ID is not dependent on the platform. If the length of the user ID is greater than that allowed, the authentication request fails with an RC2035.

This is an input field. The initial value of this field is 0.

CSCSPUIO (10-digit signed integer)

This is the offset in bytes of the user ID to be used in authentication.

The offset can be positive or negative.

This is an input field. The initial value of this field is 0.

CSCSPUIP (pointer)

This is the address of the user ID to be used in authentication.

This is an input field. The initial value of this field is the null pointer. This field is ignored if CSVER is less than CSVER5.

CSRE1 (4-byte character string)

A reserved field, required for pointer alignment on IBM i.

This is an input field. The initial value of this field is all null.

CSRS2 (8-byte character string)

A reserved field, required for pointer alignment on IBM i.

This is an input field. The initial value of this field is all null.

CSSID (4-byte character string)

Structure identifier.

The value must be:

CSSIDV

Identifier for the security parameters structure.

CSVER (10-digit signed integer)

Structure version number.

The value must be:

CSVER1

Version-1 security parameters structure.

The following constant specifies the version number of the current version:

CSVERC

Current version of security parameters structure.

This is always an input field. The initial value of this field is CSVER1.

Initial values

Table 585. Initial values of fields in MQCNO		
Field name	Name of constant	Value of constant
CSSID	CSSIDV	'CSP~'
CSVER	CSVER1	1
CSAUTHT	None	0
CSRE1	None	Nulls
CSCSPUIP	None	Null pointer
CSCSPUIO	None	0
CSCSPUIL	None	0
CSRS2	None	Nulls
CSCPPP	None	Null pointer
CSCPP0	None	0
CSCPPL	None	0

Note:

1. The symbol ~ represents a single blank character.

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D*  
D* MQCSP Structure  
D*  
D* Structure identifier  
D  CSSID                1      4      INZ('CSP ')  
D* Structure version number  
D  CSVER                5      8I 0 INZ(1)  
D* Type of authentication  
D  CSAUTHT             9     12I 0 INZ(0)  
D* Reserved  
D  CSRE1              13     16     INZ(X'00000000')  
D* Address of user ID  
D  CSCSPUIP           17     32*    INZ(*NULL)  
D* Offset of user ID  
D  CSCSPUIO           33     36I 0 INZ(0)  
D* Length of user ID  
D  CSCSPUIL           37     40I 0 INZ(0)  
D* Reserved  
D  CSRS2              41     48     INZ(X'0000000000000000')  
D* Address of password  
D  CSCPPP             49     64*    INZ(*NULL)
```

D* Offset of password			
D CSCPP0	65	68I 0	INZ(0)
D* Length of password			
D CSCPPL	69	72I 0	INZ(0)

MQCTLO - Control callback options structure

Structure specifying the control callback function.

Overview

Purpose

The MQCTLO structure is used to specify options relating to a control callbacks function.

The structure is an input and output parameter on the [MQCTL](#) call.

Version

The current version of MQCTLO is CTLV1.

Character set and encoding

Data in MQCTLO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 1017](#)
- [“Initial values” on page 1018](#)
- [“RPG declaration” on page 1018](#)

Fields

The MQCTLO structure contains the following fields; the fields are described in alphabetical order:

COCONNAREA (10 digit signed integer)

Control options structure - ConnectionArea field.

This is a field that is available for the callback function to use.

The queue manager makes no decisions based on the contents of this field and it is passed unchanged from the [CBCCONNAREA](#) field in the MQCBC structure, which is a parameter on the MQCB call.

This field is ignored for all operations other than CTLSR and CTLSW.

This is an input and output field to the callback function. The initial value of this field is a null pointer or null bytes.

COOPT (10 digit signed integer)

Options that control the action of MQCTLO.

CTLFQ

Force the MQCTLO call to fail if the queue manager or connection is in the quiescing state.

Specify GMFIQ, in the MQGMO options passed on the MQCB call, to cause notification to message consumers when they are quiescing.

CTLTHR

This option informs the system that the application requires that all message consumers, for the same connection, are called on the same thread.

Default option: If you do not need any of the options described, use the following option:

CTLNO

Use this value to indicate that no other options have been specified; all options assume their default values. CTLNO is defined to aid program documentation; it is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *COOPT* field is CTLNO.

CORSV (10 digit signed integer)

This is a reserved field. The initial value of this field is a blank character.

COSID (10 digit signed integer)

Control options structure - StrucId field.

This is the structure identifier; the value must be:

CTLSI

Identifier for Control Options structure.

This is always an input field. The initial value of this field is CTLSI.

COVER (10 digit signed integer)

Control options structure - Version field.

This is the structure version number; the value must be:

CTLV1

Version-1 Control options structure.

The following constant specifies the version number of the current version:

CTLCV

Current version of Control options structure.

This is always an input field. The initial value of this field is CTLV1.

Initial values

Table 586. Initial values of fields in MQCTLO		
Field name	Name of constant	Value of constant
<i>COSID</i>	CTLSI	'CTLO'
<i>COVER</i>	CTLV1	1
<i>COOPT</i>	CTLNO	Nulls
<i>CORSV</i>	Reserved field	
<i>COCONNAREA</i>	None	Null pointer or null bytes

RPG declaration

```
D* MQCTLO Structure
D*
D*
D* Structure identifier
D  COSID          1      4    INZ('CTLO')
D*
D* Structure version number
D  COVER          5      8I 0 INZ(1)
D*
D* Options that control the action of MQCTL
D  COOPT          9      12I 0 INZ(0)
D*
D* Reserved
D  CORSV         13      16I 0 INZ(-1)
D*
D* MQCTL Data area passed to the function
D  COCONNAREA    17      32*  INZ(*NULL)
```

MQDH - Distribution header

The MQDH structure describes the additional data that is present in a message when that message is a distribution-list message stored on a transmission queue.

Overview

Purpose: A distribution-list message is a message that is sent to multiple destination queues. The additional data consists of the MQDH structure followed by an array of MQOR records and an array of MQPMR records.

This structure is for use by specialized applications that put messages directly on transmission queues, or which remove messages from transmission queues (for example: message channel agents).

This structure should *not* be used by normal applications which simply want to put messages to distribution lists. Those applications should use the MQOD structure to define the destinations in the distribution list, and the MQPMO structure to specify message properties or receive information about the messages sent to the individual destinations.

Character set and encoding: Data in MQDH must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT for the C programming language.

The character set and encoding of the MQDH must be set into the *MDCSI* and *MDENC* fields in:

- The MQMD (if the MQDH structure is at the start of the message data), or
- The header structure that precedes the MQDH structure (all other cases).

Usage: When an application puts a message to a distribution list, and some or all of the destinations are remote, the queue manager prefixes the application message data with the MQXQH and MQDH structures, and places the message on the relevant transmission queue. The data therefore occurs in the following sequence when the message is on a transmission queue:

- MQXQH structure
- MQDH structure plus arrays of MQOR and MQPMR records
- Application message data

Depending on the destinations, more than one such message might be generated by the queue manager, and placed on different transmission queues. In this case, the MQDH structures in those messages identify different subsets of the destinations defined by the distribution list opened by the application.

An application that puts a distribution-list message directly on a transmission queue must conform to the sequence described previously, and must ensure that the MQDH structure is correct. If the MQDH structure is not valid, the queue manager may choose to fail the MQPUT or MQPUT1 call with reason code RC2135.

Messages can be stored on a queue in distribution-list form only if the queue is defined as being able to support distribution list messages (see the *DistLists* queue attribute described in “Attributes for queues” on page 1324). If an application puts a distribution-list message directly on a queue that does not support distribution lists, the queue manager splits the distribution list message into individual messages, and places those on the queue instead.

- “Fields” on page 1019
- “Initial values” on page 1022
- “RPG declaration” on page 1023

Fields

The MQDH structure contains the following fields; the fields are described in **alphabetical order**:

DHCNT (10-digit signed integer)

Number of MQOR records present.

This defines the number of destinations. A distribution list must always contain at least one destination, so *DHCNT* must always be greater than zero.

The initial value of this field is 0.

DHCSI (10-digit signed integer)

Character set identifier of data that follows the MQOR and MQPMR records.

This specifies the character set identifier of the data that follows the arrays of MQOR and MQPMR records; it does not apply to character data in the MQDH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

DHENC (10-digit signed integer)

Numeric encoding of data that follows the MQOR and MQPMR records.

This specifies the numeric encoding of the data that follows the arrays of MQOR and MQPMR records; it does not apply to numeric data in the MQDH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

DHFLG (10-digit signed integer)

General flags.

The following flag can be specified:

DHFNEW

Generate new message identifiers.

This flag indicates that a new message identifier is to be generated for each destination in the distribution list. This can be set only when there are no put-message records present, or when the records are present but they do not contain the *PRMID* field.

Using this flag defers generation of the message identifiers until the last possible moment, namely the moment when the distribution-list message is finally split into individual messages. This minimizes the amount of control information that must flow with the distribution-list message.

When an application puts a message to a distribution list, the queue manager sets DHFNEW in the MQDH it generates when both of the following are true:

- There are no put-message records provided by the application, or the records provided do not contain the *PRMID* field.
- The *MDMID* field in MQMD is MINONE, or the *PMOPT* field in MQPMO includes PMNMID

If no flags are needed, the following can be specified:

DHFNON

No flags.

This constant indicates that no flags have been specified. DHFNON is defined to aid program documentation. It is not intended that this constant is used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is DHFNON.

DHFMT (8-byte character string)

Format name of data that follows the MQOR and MQPMR records.

This specifies the format name of the data that follows the arrays of MQOD and MQPMR records (whichever occurs last).

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

The initial value of this field is FMNONE.

DHLEN (10-digit signed integer)

Length of MQDH structure plus following MQOR and MQPMR records.

This is the number of bytes from the start of the MQDH structure to the start of the message data following the arrays of MQOR and MQPMR records. The data occurs in the following sequence:

- MQDH structure
- Array of MQOR records
- Array of MQPMR records
- Message data

The arrays of MQOR and MQPMR records are addressed by offsets contained within the MQDH structure. If these offsets result in unused bytes between one or more of the MQDH structure, the arrays of records, and the message data, those unused bytes must be included in the value of *DHLEN*, but the content of those bytes is not preserved by the queue manager. It is valid for the array of MQPMR records to precede the array of MQOR records.

The initial value of this field is 0.

DHORO (10-digit signed integer)

Offset of first MQOR record from start of MQDH.

This field gives the offset in bytes of the first record in the array of MQOR object records containing the names of the destination queues. There are *DHCNT* records in this array. These records (plus any bytes skipped between the first object record and the previous field) are included in the length given by the *DHLEN* field.

A distribution list must always contain at least one destination, so *DHORO* must always be greater than zero.

The initial value of this field is 0.

DHPRF (10-digit signed integer)

Flags indicating which MQPMR fields are present.

Zero or more of the following flags can be specified:

PFMID

Message-identifier field is present.

PFCID

Correlation-identifier field is present.

PFGID

Group-identifier field is present.

PFFB

Feedback field is present.

PFACC

Accounting-token field is present.

If no MQPMR fields are present, the following can be specified:

PFNONE

No put-message record fields are present.

PFNONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is PFNONE.

DHPRO (10-digit signed integer)

Offset of first MQPMR record from start of MQDH.

This field gives the offset in bytes of the first record in the array of MQPMR put message records containing the message properties. If present, there are *DHCNT* records in this array. These records (plus any bytes skipped between the first put message record and the previous field) are included in the length given by the *DHLEN* field.

Put message records are optional; if no records are provided, *DHPRO* is zero, and *DHPRF* has the value PFNONE.

The initial value of this field is 0.

DHSID (4-byte character string)

Structure identifier.

The value must be:

DHSIDV

Identifier for distribution header structure.

The initial value of this field is DHSIDV.

DHVER (10-digit signed integer)

Structure version number.

The value must be:

DHVER1

Version number for distribution header structure.

The following constant specifies the version number of the current version:

DHVERC

Current version of distribution header structure.

The initial value of this field is DHVER1.

Initial values

Table 587. Initial values of fields in MQDH		
Field name	Name of constant	Value of constant
<i>DHSID</i>	DHSIDV	'DH--'
<i>DHVER</i>	DHVER1	1
<i>DHLEN</i>	None	0
<i>DHENC</i>	None	0
<i>DHCSI</i>	CSUNDF	0
<i>DHFMT</i>	FMNONE	Blanks

Table 587. Initial values of fields in MQDH (continued)

Field name	Name of constant	Value of constant
DHFLG	DHFNON	0
DHPRF	PFNONE	0
DHCNT	None	0
DHORO	None	0
DHPRO	None	0

Notes:

1. The symbol – represents a single blank character.

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQDH Structure
D*
D* Structure identifier
D  DHSID          1          4      INZ('DH ')
D* Structure version number
D  DHVER          5          8I 0 INZ(1)
D* Length of MQDH structure plusfollowing MQOR and MQPMR records
D  DHLEN          9          12I 0 INZ(0)
D* Numeric encoding of data that followsthe MQOR and MQPMR records
D  DHENC         13          16I 0 INZ(0)
D* Character set identifier of data thatfollows the MQOR and MQPMR
D* records
D  DHCSI         17          20I 0 INZ(0)
D* Format name of data that follows theMQOR and MQPMR records
D  DHFMT         21          28      INZ(' ')
D* General flags
D  DHFLG         29          32I 0 INZ(0)
D* Flags indicating which MQPMR fieldsare present
D  DHPRF         33          36I 0 INZ(0)
D* Number of MQOR records present
D  DHCNT         37          40I 0 INZ(0)
D* Offset of first MQOR record from startof MQDH
D  DHORO         41          44I 0 INZ(0)
D* Offset of first MQPMR record fromstart of MQDH
D  DHPRO         45          48I 0 INZ(0)

```

MQDLH - Dead-letter header

Overview

Purpose

The MQDLH structure describes the information that prefixes the application message data of messages on the dead-letter (undelivered-message) queue. A message can arrive on the dead-letter queue because the queue manager or message channel agent redirected it to the queue. An application might put the message directly on the queue.

Format name

FMDLH

Character set and encoding

The MQDLH might be at the start of the application message data. If so, the fields in the MQDLH structure are in the character set and encoding given by the MDCSI and MDENC fields. If not, the character set and encoding are set by the MDCSI and MDENC fields in the header structure that precedes the MQDLH.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

Usage

Applications that put messages directly on the dead-letter queue must prefix the message data with an MQDLH structure, and initialize the fields with appropriate values. However, the queue manager does not require that an MQDLH structure is present, or that valid values are specified for the fields.

If a message is too long to put on the dead-letter queue, the application must consider doing one of the following things:

- Truncate the message data to fit on the dead-letter queue.
- Record the message on auxiliary storage and place an exception report message on the dead-letter queue indicating the message is too long.
- Discard the message and return an error to its originator. If the message is a critical message. Discard the message only if it is known that the originator still has a copy of the message. For example, a message received by a message channel agent from a communication channel.

Which of the choices is appropriate depends on the design of the application.

The queue manager performs special processing when a message which is a segment is put with an MQDLH structure at the front. See the description of the MQMDE structure for further details.

- [“Putting messages on the dead-letter queue” on page 1024](#)
- [“Getting messages from the dead-letter queue” on page 1025](#)
- [“Fields” on page 1025](#)
- [“Initial values” on page 1028](#)
- [“RPG declaration” on page 1029](#)

Putting messages on the dead-letter queue

If a message is put on the dead-letter queue, the MQMD structure used for the MQPUT or MQPUT1 call must be identical to the MQMD associated with the message. The MQMD is typically the one returned by the MQGET call, except for the following cases:

- The MDCSI and MDENC fields must be set to whatever character set and encoding are used for fields in the MQDLH structure.
- The MDFMT field must be set to FMDLH to indicate that the data begins with an MQDLH structure.
- The context fields, MDACC, MDAID, MDAOD, MDPAN, MDPAT, MDPD, MDPT, and MDUID must be set by using a context option appropriate to the circumstances:
 - An application putting on the dead-letter queue a message that is not related to any preceding message must use the PMDEFC option. The PMDEFC option causes the queue manager to set all of the context fields in the message descriptor to their default values.
 - A server application putting on the dead-letter queue a message it received must use the PMPASA option, in order to preserve the original context information.
 - A server application putting on the dead-letter queue a reply to message it received must use the PMPASI option. The PMPASI option preserves the identity information but sets the origin information to be that of the server application.
 - A message channel agent putting on the dead-letter queue a message it received from its communication channel must use the PMSETA option. The PMSETA option preserves the original context information.

In the MQDLH structure itself, the fields must be set as follows:

- The DLCSI, DLENC, and *DLFMT* fields must be set to the values that describe the data that follows the MQDLH structure. These values are typically the values from the original message descriptor.
- The context fields DLPAT, DLPAN, DLPD, and DLPT must be set to values appropriate to the application that is putting the message on the dead-letter queue. These values are not related to the original message.
- Other fields must be set as appropriate.

The application must ensure that all fields have valid values, and that character fields are padded with blanks to the defined length of the field. The character data must not be terminated prematurely by using a null character. The queue manager does not convert the null and subsequent characters to blanks in the MQDLH structure.

Getting messages from the dead-letter queue

Applications that get messages from the dead-letter queue must verify that the messages begin with an MQDLH structure. The application can determine whether an MQDLH structure is present by examining the MDFMT field in the message descriptor MQMD. If the field has the value FMDLH, the message data begins with an MQDLH structure. Messages on the dead-letter queue might be truncated if they were originally too long for the queue they were intended for.

Fields

The MQDLH structure contains the following fields; the fields are described in alphabetical order:

DLCSI (10-digit signed integer)

Character set identifier of data that follows MQDLH.

DLCSI specifies the character set identifier of the data that follows the MQDLH structure. The data is typically from the original message. It does not apply to character data in the MQDLH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data following this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the MDPAT field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

DLDM (48-byte character string)

Name of original destination queue manager.

This is the name of the queue manager that was the original destination for the message.

The length of this field is given by LNQM. The initial value of this field is 48 blank characters.

DLDQ (48-byte character string)

Name of original destination queue.

This is the name of the message queue that was the original destination for the message.

The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

DLENC (10-digit signed integer)

Numeric encoding of data that follows MQDLH.

DLENC specifies the numeric encoding of the data that follows the MQDLH structure. The data is typically from the original message. It does not apply to numeric data in the MQDLH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

DLFMT (8-byte character string)

Format name of data that follows MQDLH.

This specifies the format name of the data that follows the MQDLH structure (typically the data from the original message).

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as the rules for the MDFMT field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

DLPAN (28-byte character string)

Name of application that put message on dead-letter (undelivered-message) queue.

The format of the name depends on the DLPAT field. See the description of the MDPAN field in [“MQMD - Message descriptor”](#) on page 1068.

If it is the queue manager that redirects the message to the dead-letter queue, DLPAN contains the first 28 characters of the queue manager name. The name is padded with blanks if necessary.

The length of this field is given by LNPAN. The initial value of this field is 28 blank characters.

DLPAT (10-digit signed integer)

Type of application that put message on dead-letter (undelivered-message) queue.

This field has the same meaning as the MDPAT field in the message descriptor MQMD (see [“MQMD - Message descriptor”](#) on page 1068 for details).

If it is the queue manager that redirects the message to the dead-letter queue, DLPAT has the value ATQM.

The initial value of this field is 0.

DLPD (8-byte character string)

Date when message was put on dead-letter (undelivered-message) queue.

The format used for the date when this field is generated by the queue manager is:

- YYYYMMDD

where the characters represent:

YYYY

year (four numeric digits)

MM

month of year (01 through 12)

DD

day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the DLPD and DLPT fields, subject to the system clock being set accurately to GMT.

The length of this field is given by LNPDAT. The initial value of this field is eight blank characters.

DLPT (8-byte character string)

Time when message was put on the dead-letter (undelivered-message) queue.

The format used for the time when this field is generated by the queue manager is:

- HHMMSSTH

where the characters represent (in order):

HH

hours (00 through 23)

MM

minutes (00 through 59)

SS

seconds (00 through 59; see note later in this topic)

T

tenths of a second (0 through 9)

H

hundredths of a second (0 through 9)

Note: If the system clock is synchronized to an accurate time standard, it is possible for 60 or 61 to be returned for the seconds in DLPT. The extra second occurs when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the DLPD and DLPT fields, subject to the system clock being set accurately to GMT.

The length of this field is given by LNPTIM. The initial value of this field is eight blank characters.

DLREA (10-digit signed integer)

Reason message arrived on dead-letter (undelivered-message) queue.

This identifies the reason why the message was placed on the dead-letter queue instead of on the original destination queue. It must be one of the FB* or RC* values (for example, RC2053). See the description of the *MDFB* field in [“MQMD - Message descriptor” on page 1068](#) for details of the common FB* values that can occur.

If the value is in the range FBIFST through FBILST, the actual IMS error code can be determined by subtracting FBIERR from the value of the *DLREA* field.

Some FB* values occur only in this field. They relate to repository messages, trigger messages, or transmission-queue messages that are transferred to the dead-letter queue. These values are:

FBABEG

Application cannot be started.

An application processing a trigger message was unable to start the application named in the TMAI field of the trigger message; see [“MQTM - Trigger message” on page 1191](#).

FBATYP

Application type error.

An application processing a trigger message was unable to start the application because the TMAI field of the trigger message is not valid; see [“MQTM - Trigger message” on page 1191](#).

FBBOCD

Cluster-receiver channel deleted.

The message was on a cluster transmission queue intended for a cluster queue that was opened with the FBIERR option. The remote cluster-receiver channel to be used to transmit the message to the destination queue was deleted before the message could be sent. Because FBIERR was specified, only the channel selected when the queue was opened can be used to transmit the message. As this channel is not longer available, the message was placed on the dead-letter queue.

FBNARM

Message is not a repository message.

FBSBCX

Message stopped by channel auto-definition exit.

FBSBMX

Message stopped by channel message exit.

FBTM

MQTM structure not valid or missing.

The MDFMT field in MQMD specifies FMTM, but the message does not begin with a valid MQTM structure. For example, the *TMSID* mnemonic eye-catcher might not be valid. The *TMVER* might

not be recognized. The length of the trigger message might be insufficient to contain the MQTM structure.

FBXQME

Message on transmission queue not in correct format.

A message channel agent found that a message on the transmission queue is not in the correct format. The message channel agent puts the message on the dead-letter queue using this feedback code.

The initial value of this field is RCNONE.

DLSID (4-byte character string)

Structure identifier.

The value must be:

DLSIDV

Identifier for dead-letter header structure.

The initial value of this field is DLSIDV.

DLVER (10-digit signed integer)

Structure version number.

The value must be:

DLVER1

Version number for dead-letter header structure.

The following constant specifies the version number of the current version:

DLVERC

Current version of dead-letter header structure.

The initial value of this field is DLVER1.

Initial values

Table 588. Initial values of fields in MQDLH.		
Lists the names of the MQDLH constants and their values.		
Field name	Name of constant	Value of constant
DLSID	DLSIDV	'DLH~'
DLVER	DLVER1	1
DLREA	RCNONE	0
DLDQ	None	Blanks
DLDM	None	Blanks
DLENC	None	0
DLCSI	CSUNDF	0
DLFMT	FMNONE	Blanks
DLPAT	None	0
DLPAN	None	Blanks
DLPD	None	Blanks
DLPT	None	Blanks

Notes:

1. The symbol – represents a single blank character.

RPG declaration

```
D*..1....2....3....4....5....6....7..
D* MQDLH Structure
D*
D* Structure identifier
D  DLSID          1      4      INZ('DLH ')
D* Structure version number
D  DLVER          5      8I 0 INZ(1)
D* Reason message arrived on dead-letter(undelivered-message) queue
D  DLREA          9     12I 0 INZ(0)
D* Name of original destination queue
D  DLDQ          13     60      INZ
D* Name of original destination queue manager
D  DLDM          61     108      INZ
D* Numeric encoding of data that followsMQDLH
D  DLENC         109    112I 0 INZ(0)
D* Character set identifier of data that follows MQDLH
D  DLCSI         113    116I 0 INZ(0)
D* Format name of data that followsMQDLH
D  DLFMT         117    124      INZ(' ')
D* Type of application that put message on dead-letter
D* (undelivered-message)queue
D  DLPAT         125    128I 0 INZ(0)
D* Name of application that put message on dead-letter
D* (undelivered-message)queue
D  DLPAN         129    156      INZ
D* Date when message was put on dead-letter (undelivered-message)queue
D  DLPD          157    164      INZ
D* Time when message was put on the dead-letter (undelivered-message)queue
D  DLPT          165    172      INZ
```

MQDMHO - Delete message handle options

The **MQDMHO** structure allows applications to specify options that control how message handles are deleted.

Overview

Purpose: The structure is an input parameter on the **MQDLTMH** call.

Character set and encoding: Data in **MQDMHO** must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1029](#)
- [“Initial values” on page 1030](#)
- [“RPG declaration” on page 1030](#)

Fields

The **MQDMHO** structure contains the following fields; the fields are described in **alphabetical order**:

DMOPT (10-digit signed integer)

The value must be:

DMNONE

No options specified.

This is always an input field. The initial value of this field is **DMNONE**.

DMSID (10-digit signed integer)

This is the structure identifier; the value must be:

DMSIDV

Identifier for delete message handle options structure.

This is always an input field. The initial value of this field is **DMSIDV**.

DMVER (10-digit signed integer)

This is the structure version number; the value must be:

DMVER1

Version-1 delete message handle options structure.

The following constant specifies the version number of the current version:

DMVERC

Current version of delete message handle options structure.

This is always an input field. The initial value of this field is **DMVER1**.

Initial values

Table 589. Initial values of fields in MQDMHO		
Field name	Name of constant	Value of constant
DMSID	DMSIDV	' DMHO '
DMVER	DMVER1	1
DMOPT	DMNONE	0

RPG declaration

```
D* MQDMHO Structure
D*
D*
D* Structure identifier
D  DMSID              1      4      INZ('DMHO')
D*
D* Structure version number
D  DMVER              5      8I 0 INZ(1)
D*
D* Options that control the action of MQDLTMH
D  DMOPT              9      12I 0 INZ(0)
```

MQDMPO - Delete message property options

Structure defining the delete message property options.

Overview

Purpose: The MQDMPO structure allows applications to specify options that control how properties of messages are deleted. The structure is an input parameter on the MQDLTMP call.

Character set and encoding: Data in MQDMPO must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1031](#)
- [“Initial values” on page 1032](#)
- [“RPG declaration” on page 1032](#)

Fields

The MQDMPO structure contains the following fields; the fields are described in alphabetic order:

DPOPT (10-digit signed integer)

Delete message property options structure - DPOPT field.

Location options: The following options relate to the relative location of the property compared to the property cursor.

DPDELF

Deletes the first property that matches the specified name.

DPDELC

Deletes the property pointed to by the property cursor; that is the property that was last inquired by using either the IPINQF or the IPINQN option.

The property cursor is reset when the message handle is reused. It is also reset when the message handle is specified in the *HMSG* field of the MQGMO on an MQGET call, or MQPMO structure on an MQPUT call.

The property cursor is reset when the message handle is reused, or when the message handle is specified in the *HMSG* field of the MQGMO structure on an MQGET structure on an MQGET call or MQPMO structure on an MQPUT call.

The call fails with completion code CCFAIL and reason RC2471 if this option is used when the property cursor has not yet been established. It also fails with these codes if the property pointed to by the property cursor has already been deleted..

If neither of these options is required, the following option can be used:

DPNONE

No options specified.

The initial value of this input field is DPDELF.

DPSID (10-digit signed integer)

Delete message property options structure - DPSID field.

This is the structure identifier. The value must be:

DPSIDV

Identifier for delete message property options structure.

This field is always an input field. The initial value of this field is DPSIDV.

DPVER (10-digit signed integer)

Delete message property options structure - DPVER field.

This is the structure version number. The value must be:

DPVER1

Version number for delete message property options structure.

The following constant specifies the version number of the current version:

DPVERC

Current version of delete message property options structure.

This field is always an input field. The initial value of this field is DPVER1.

Initial values

Table 590. Initial values of fields in MQDPMO		
Field name	Name of constant	Value of constant
DPSID	DPSIDV	' DMP0 '
DPVER	DPVER1	1
DPOPT	Options that control the action of MQDLTMP	DPNONE

RPG declaration

```
D* MQDPMO Structure
D*
D*
D* Structure identifier
D  DPSID              1      4      INZ(' DMP0 ')
D*
D* Structure version number
D  DPVER              5      8I 0  INZ(1)
D*
** Options that control the action of
D* MQDLTMP
D  DPOPT              9      12I 0  INZ(0)
```

MQEPH - Embedded PCF header

Overview

Purpose

The MQEPH structure describes the additional data that is present in a message when that message is a programmable command format (PCF) message. The *EPPFH* field defines the PCF parameters that follow this structure and this allows you to follow the PCF message data with other headers.

Format name

EPFMT

Character set and encoding

Data in MQEPH must be in the character set and encoding of the local queue manager; this is given by the *CCSID* queue-manager attribute.

Set the character set and encoding of the MQEPH into the *MDCSI* and *MDENC* fields in:

- The MQMD (if the MQEPH structure is at the start of the message data), or
- The header structure that precedes the MQEPH structure (all other cases).

Usage

You cannot use MQEPH structures to send commands to the command server or any other queue manager PCF-accepting server.

Similarly, the command server or any other queue manager PCF-accepting server do not generate responses or events containing MQEPH structures.

- [“Fields” on page 1032](#)
- [“Initial values” on page 1034](#)
- [“RPG declaration” on page 1035](#)

Fields

The MQEPH structure contains the following fields; the fields are described in **alphabetical order**:

EPCSI (10-digit signed integer)

This is the character set identifier of the data that follows the MQEPH structure and the associated PCF parameters; it does not apply to character data in the MQEPH structure itself.

The initial value of this field is EPCUND.

EPENC (10-digit signed integer)

This is the numeric encoding of the data that follows the MQEPH structure and the associated PCF parameters; it does not apply to character data in the MQEPH structure itself.

The initial value of this field is 0.

EPFLG (10-digit signed integer)

The following values are available:

EPNONE

No flags have been specified. *MDCSI* EPNONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

EPCSEM

The character set of the parameters containing character data is specified individually within the *CCSID* field in each structure. The character set of the *EPSID* and *EPFMT* fields are defined by the *CCSID* in the header structure that precedes the MQEPH structure, or by the *MDCSI* field in the MQMD if the MQEPH is at the start of the message.

The initial value of this field is EPNONE.

EPFMT (8-byte character string)

This is the format name of the data that follows the MQEPH structure and the associated PCF parameters.

The initial value of this field is EPFMNO.

EPLEN (10-digit signed integer)

This is the amount of data preceding the next header structure. It includes:

- The length of the MQEPH header
- The length of all PCF parameters following the header
- Any blank padding following those parameters

EPLEN must be a multiple of 4.

The fixed-length part of the structure is defined by EPSTLF.

The initial value of this field is 68.

EPPCFH (MQCFH)

This is the programmable command format (PCF) header, defining the PCF parameters that follow the MQEPH structure. This enables you to follow the PCF message data with other headers.

The PCF header is initially defined with the following values:

Table 591. Initial values of fields in EPPCFH		
Field name	Name of constant	Value of constant
EP3TYP	CFTNON	0
EP3LEN	FHLENV	36
EP3VER	FHVER3	3
EP3CMD	CMNONE	0

<i>Table 591. Initial values of fields in EPPCFH (continued)</i>		
Field name	Name of constant	Value of constant
EP3SEQ	None	1
EP3CTL	CFCLST	1
EEP3CC	CCOK	0
EP3REA	RCNONE	0
EP3CNT	None	0

The application must change EP3TYP from CFTNON to a valid structure type for the use it is making of the embedded PCF header.

EPSID (4-byte character string)

The value must be:

EPSTID

Identifier for the Embedded PCF header structure.

The initial value of this field is EPSTID.

EPVER (10-digit signed integer)

The value can be:

EPVER1

Version number for embedded PCF header structure.

The following constant specifies the version number of the current version:

EPVER3

Current version of embedded PCF header structure.

The initial value of this field is EPVER3.

Initial values

<i>Table 592. Initial values of fields in MQEPH</i>		
Field name	Name of constant	Value of constant
EPSID	EPSTID	'EP--'
EPVER	EPVER1	1
EPLEN	EPSTLF	68
EPENC	None	0
EPCSI	EPCUND	0
EPFMT	EPFMNO	Blanks
EPFLG	EPNONE	0
EPPCFH	Names and values as defined in Table 591 on page 1033	0

Note:

1. The symbol – represents a single blank character.

RPG declaration

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..  
D* MQEPH Structure  
D*  
D* Structure identifier  
D EPSID 1 4  
D* Structure version number  
D EPVER 5 8I 0  
D* Total length of MQEPH including MQCFHand parameter structures  
D* that follow  
D EPLEN 9 12I 0  
D* Numeric encoding of data that follows last PCF parameter structure  
D EPENC 13 16I 0  
D* Character set identifier of data that follows last PCF parameter  
D* structure  
D EPCSI 17 20I 0  
D* Format name of data that follows last PCF parameter structure  
D EPFMT 21 28  
D* Flags  
D EPFLG 29 32I 0  
D* Programmable Command Format Header  
D EP3TYP 33 36I 0  
D EP3LEN 37 40I 0  
D EP3VER 41 44I 0  
D EP3CMD 45 48I 0  
D EP3SEQ 49 52I 0  
D EP3CTL 53 56I 0  
D EP3CC 57 60I 0  
D EP3REA 61 64I 0  
D EP3CNT 65 68I 0
```

MQGMO - Get-message options

The MQGMO structure allows the application to specify options that control how messages are removed from queues.

Overview

Purpose

The structure is an input/output parameter on the MQGET call.

Version

The current version of MQGMO is GMVER4. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQGMO that is supported by the environment, but with the initial value of the *GMVER* field set to GMVER1. To use fields that are not present in the version-1 structure, the application must set the *GMVER* field to the version number of the version required.

Character set and encoding

Data in MQGMO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 1035](#)
- [“Initial values” on page 1055](#)
- [“RPG declaration” on page 1055](#)

Fields

The MQGMO structure contains the following fields; the fields are described in alphabetical order:

GMGST (1 byte character string)

Flag indicating whether message retrieved is in a group.

It has one of the following values:

GSNIG

Message is not in a group.

GSMIG

Message is in a group, but is not the last in the group.

GSLMIG

Message is the last in the group.

This value is also the value returned if the group consists of only one message.

This field is an output field. The initial value of this field is GSNIG. This field is ignored if *GMVER* is less than GMVER2.

GMMH (10 digit signed integer)

Message Handle

If the GMPRAQ option is specified and the PRPCTL queue attribute is not set to PRPRFH then this is the handle to a message which is populated with the properties of the message being retrieved from the queue. The handle is created by an MQCRTMH call. Any properties already associated with the handle are cleared before retrieving a message.

The following value can also be specified:

MQHM_NONE

No message handle supplied.

No message descriptor is required on the MQGET call if a valid message handle is supplied and used on output to contain the message properties, the message descriptor associated with the message handle is used for input fields.

If a message descriptor is specified on the MQGET call, it always takes precedence over the message descriptor associated with a message handle.

If GMPRRF is specified, or the GMPRAQ is specified and the PRPCTL queue attribute is PRPRFH then the call fails with reason code RC2026 when no message descriptor parameter is specified.

On return from the MQGET call, the properties and message descriptor associated with this message handle are updated to reflect the state of the message retrieved (as well as the message descriptor if one was supplied on the MQGET call). The properties of the message can then be inquired using the MQINQMP call.

Except for message descriptor extensions, when present, a property that can be inquired with the MQINQMP call is not contained in the message data; if the message on the queue contained properties in the message data these are removed from the message data before the data is returned to the application.

If no message handle is provided or Version is less than GMVER4 then you must supply a valid message descriptor on the MQGET call. Any message properties (except those properties contained in the message descriptor) are returned in the message data subject to the value of the property options in the MQGMO structure and the PRPCTL queue attribute.

This field is an always an input field. The initial value of this field is HMNONE. This field is ignored if *GMVER* is less than GMVER4.

GMMO (10 digit signed integer)

Options controlling selection criteria used for MQGET.

These options allow the application to choose which fields in the *MSGDSC* parameter is used to select the message returned by the MQGET call. The application sets the required options in this field, and then sets the corresponding fields in the *MSGDSC* parameter to the values required for those fields. Only messages that have those values in the MQMD for the message are candidates for retrieval using that *MSGDSC* parameter on the MQGET call. Fields for which the corresponding match option is not specified are ignored when selecting the message to be returned. If no selection criteria are to be used on the MQGET call (that is, any message is acceptable), *GMMO* should be set to MONONE.

If GMLOGO is specified, only certain messages are eligible for return by the next MQGET call:

- If there is no current group or logical message, only messages that have *MDSEQ* equal to 1 and *MDOFF* equal to 0 are eligible for return. In this situation, one or more of the following options can be used to select which of the eligible messages is the one returned:
 - MOMSGI
 - MOCORI
 - MOGRPI
- If there is a current group or logical message, only the next message in the group or next segment in the logical message is eligible for return, and this cannot be altered by specifying MO* options.

In both cases, match options which are not applicable can still be specified, but the value of the relevant field in the *MSGDSC* parameter must match the value of the corresponding field in the message to be returned; the call fails with reason code RC2247 if this condition is not satisfied.

GMMO is ignored if either *GMMUC* or *GMBRWC* is specified.

One or more of the following options can be specified:

MOMSGI

Retrieve message with specified message identifier.

This option specifies that the message to be retrieved must have a message identifier that matches the value of the *MDMID* field in the *MSGDSC* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the correlation identifier).

If this option is not specified, the *MDMID* field in the *MSGDSC* parameter is ignored, and any message identifier matches.

Note: The message identifier MINONE is a special value that matches any message identifier in the MQMD for the message. Therefore, specifying MOMSGI with MINONE is the same as not specifying MOMSGI.

MOCORI

Retrieve message with specified correlation identifier.

This option specifies that the message to be retrieved must have a correlation identifier that matches the value of the *MDCID* field in the *MSGDSC* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the message identifier).

If this option is not specified, the *MDCID* field in the *MSGDSC* parameter is ignored, and any correlation identifier matches.

Note: The correlation identifier CINONE is a special value that matches any correlation identifier in the MQMD for the message. Therefore, specifying MOCORI with CINONE is the same as not specifying MOCORI.

MOGRPI

Retrieve message with specified group identifier.

This option specifies that the message to be retrieved must have a group identifier that matches the value of the *MDGID* field in the *MSGDSC* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the correlation identifier).

If this option is not specified, the *MDGID* field in the *MSGDSC* parameter is ignored, and any group identifier matches.

Note: The group identifier GINONE is a special value that matches any group identifier in the MQMD for the message. Therefore, specifying MOGRPI with GINONE is the same as not specifying MOGRPI.

MOSEQN

Retrieve message with specified message sequence number.

This option specifies that the message to be retrieved must have a message sequence number that matches the value of the *MDSEQ* field in the *MSGDSC* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the group identifier).

If this option is not specified, the *MDSEQ* field in the *MSGDSC* parameter is ignored, and any message sequence number matches.

MOFFS

Retrieve message with specified offset.

This option specifies that the message to be retrieved must have an offset that matches the value of the *MDOFF* field in the *MSGDSC* parameter of the MQGET call. This match is in addition to any other matches that might apply (for example, the message sequence number).

If this option is not specified, the *MDOFF* field in the *MSGDSC* parameter is ignored, and any offset matches.

If none of the options described is specified, the following option can be used:

MONONE

No matches.

This option specifies that no matches are to be used in selecting the message to be returned; therefore, all messages on the queue are eligible for retrieval (but subject to control by the GMAMSA, GMASGA, and GMCMPM options).

MONONE is defined to aid program documentation. It is not intended that this option is used with any other MO* option, but as its value is zero, such use cannot be detected.

This field is an input field. The initial value of this field is MOMSGI with MOCORI. This field is ignored if *GMVER* is less than *GMVER2*.

Note: The initial value of the *GMMO* field is defined for compatibility with earlier version queue managers. However, when reading a series of messages from a queue without using selection criteria, this initial value requires the application to reset the *MDMID* and *MDCID* fields to MINONE and CINONE before each MQGET call. The need to reset *MDMID* and *MDCID* can be avoided by setting *GMVER* to *GMVER2*, and *GMMO* to MONONE.

GMOPT (10 digit signed integer)

Options that control the action of MQGET.

Zero or more of the following described options can be specified. If more than one is required the values can be added (do not add the same constant more than once). Combinations of options that are not valid are noted; all other combinations are valid.

Wait options: The following options relate to waiting for messages to arrive on the queue:

GMWT

Wait for message to arrive.

The application is to wait until a suitable message arrives. The maximum time the application waits is specified in *GMWT*.

If MQGET requests are inhibited, or MQGET requests become inhibited while waiting, the wait is canceled and the call completes with CCFAIL and reason code RC2016, regardless of whether there are suitable messages on the queue.

This option can be used with the GMBRWF or GMBRWN options.

If several applications are waiting on the same shared queue, the application, or applications, that are activated when a suitable message arrives are described later in this section.

Note: In the following description, a browse MQGET call is one which specifies one of the browse options, but not GMLK; an MQGET call specifying the GMLK option is treated as a nonbrowse call.

- If one or more nonbrowse MQGET calls is waiting, but no browse MQGET calls are waiting, one is activated.

- If one or more browse MQGET calls is waiting, but no nonbrowse MQGET calls are waiting, all are activated.
- If one or more nonbrowse MQGET calls, and one or more browse MQGET calls are waiting, one nonbrowse MQGET call is activated, and none, some, or all the browse MQGET calls. (The number of browse MQGET calls activated cannot be predicted, because it depends on the scheduling considerations of the operating system, and other factors.)

If more than one nonbrowse MQGET call is waiting on the same queue, only one is activated; in this situation the queue manager attempts to give priority to waiting nonbrowse calls in the following order:

1. Specific get-wait requests that can be satisfied only by certain messages, for example, ones with a specific *MDMID* or *MDCID* (or both).
2. General get-wait requests that can be satisfied by any message.

The following points must be noted:

- Within the first category, no additional priority is given to more specific get-wait requests, for example those that specify both *MDMID* and *MDCID*.
- Within either category, it cannot be predicted which application is selected. In particular, the application waiting longest is not necessarily the one selected.
- Path length, and priority-scheduling considerations of the operating system, can mean that a waiting application of lower operating system priority than expected retrieves the message.
- It might also happen that an application that is not waiting retrieves the message in preference to one that is.

GMWT is ignored if specified with GMBRWC or GMMUC; no error is raised.

GMNWT

Return immediately if no suitable message.

The application is not to wait if no suitable message is available. This is the opposite of the GMWT option, and is defined to aid program documentation. It is the default if neither is specified.

GMFIQ

Fail if queue manager is quiescing.

This option forces the MQGET call to fail if the queue manager is in the quiescing state.

If this option is specified together with GMWT, and the wait is outstanding at the time the queue manager enters the quiescing state:

- The wait is canceled and the call returns completion code CCFAIL with reason code RC2161 .

If GMFIQ is not specified and the queue manager enters the quiescing state, the wait is not canceled.

Syncpoint options: The following options relate to the participation of the MQGET call within a unit of work:

GMSYP

Get message with syncpoint control.

The request is to operate within the normal unit-of-work protocols. The message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

If this option or GMNSYP is not specified, the get request is not within a unit of work.

This option is not valid with any of the following options:

- GMBRWF
- GMBRWC
- GMBRWN

- GMLK
- GMNSYP
- GMPSYP
- GMUNLK

GMPSYP

Get message with syncpoint control if message is persistent.

The request is to operate within the normal unit-of-work protocols, but only if the message retrieved is persistent. A persistent message has the value PEPER in the *MDPER* field in MQMD.

- If the message is persistent, the queue manager processes the call as though the application had specified GMSYP.
- If the message is not persistent, the queue manager processes the call as though the application had specified GMNSYP (see the following section for details).

This option is not valid with any of the following options:

- GMBRWF
- GMBRWC
- GMBRWN
- GMCMPM
- GMNSYP
- GMSYP
- GMUNLK

GMNSYP

Get message without syncpoint control.

The request is to operate outside the normal unit-of-work protocols. The message is deleted from the queue immediately (unless this is a browse request). The message cannot be made available again by backing out the unit of work.

This option is assumed if GMBRWF or GMBRWN is specified.

If this option and GMSYP are not specified, the get request is not within a unit of work.

This option is not valid with any of the following options:

- GMSYP
- GMPSYP

Browse options: The following options relate to browsing messages on the queue:

GMBRWF

Browse from start of queue.

When a queue is opened with the OOBWF option, a browse cursor is established, positioned logically before the first message on the queue. Subsequent MQGET calls specifying the GMBRWF, GMBRWN, or GMBRWC option can be used to retrieve messages from the queue nondestructively. The browse cursor marks the position, within the messages on the queue, from which the next MQGET call with GMBRWN searches for a suitable message.

An MQGET call with GMBRWF causes the previous position of the browse cursor to be ignored. The first message on the queue that satisfies the conditions specified in the message descriptor is retrieved. The message remains on the queue, and the browse cursor is positioned on this message.

After this call, the browse cursor is positioned on the message that has been returned. If the message is removed from the queue before the next MQGET call with GMBRWN is issued, the browse cursor remains at the position in the queue that the message occupied, even though that position is now empty.

The GMMUC option can then be used with a nonbrowse MQGET call if required, to remove the message from the queue.

The browse cursor is not moved by a nonbrowse MQGET call using the same *HOB*J handle. Nor is it moved by a browse MQGET call that returns a completion code of CCFAIL, or a reason code of RC2080 .

The GMLK option can be specified together with this option, to cause the message that is browsed to be locked.

GMBRWF can be specified with any valid combination of the GM* and MO* options that control the processing of messages in groups and segments of logical messages.

If GMLOGO is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When GMBRWF is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using GMBRWN must browse the queue in the same order as the most recent call that specified GMBRWF for the queue handle.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue, is separate from the group and segment information that the queue manager retains for MQGET calls that remove messages from the queue. When GMBRWF is specified, the queue manager ignores the group and segment information for browsing, and scans the queue as though there were no current group and no current logical message. If the MQGET call is successful (completion code CCOK or CCWARN), the group and segment information for browsing is set to that of the message returned; if the call fails, the group and segment information remains the same as it was before the call.

This option is not valid with any of the following options:

- GMBRWC
- GMBRWN
- GMMUC
- GMSYP
- GMPSPY
- GMUNLK

It is also an error if the queue was not opened for browse.

GMBRWN

Browse from current position in queue.

The browse cursor is advanced to the next message on the queue that satisfies the selection criteria specified on the MQGET call. The message is returned to the application, but remains on the queue.

After a queue has been opened for browse, the first browse call using the handle has the same effect whether it specifies the GMBRWF or GMBRWN option.

If the message is removed from the queue before the next MQGET call with GMBRWN is issued, the browse cursor logically remains at the position in the queue that the message occupied, even though that position is now empty.

Messages are stored on the queue in one of two ways:

- FIFO within priority (MSPRIO), or
- FIFO regardless of priority (MSFIFO)

The *MsgDeliverySequence* queue attribute indicates which method applies (see [“Attributes for queues”](#) on page 1324 for details).

If the queue has a *MsgDeliverySequence* of MSPRIO, and a message arrives on the queue that is of a higher priority than the one currently pointed to by the browse cursor, that message is

not found during the current sweep of the queue using GMBRWN. It can only be found after the browse cursor has been reset with GMBRWF (or by reopening the queue).

The GMMUC option can later be used with a nonbrowse MQGET call if required, to remove the message from the queue.

The browse cursor is not moved by nonbrowse MQGET calls using the same *HOBJ* handle.

The GMLK option can be specified together with this option, to cause the message that is browsed to be locked.

GMBRWN can be specified with any valid combination of the GM* and MO* options that control the processing of messages in groups and segments of logical messages.

If GMLOGO is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When GMBRWF is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using GMBRWN must browse the queue in the same order as the most recent call that specified GMBRWF for the queue handle. The call fails with reason code RC2259 if this condition is not satisfied.

Note: Special care is needed if an MQGET call is used to browse beyond the end of a message group (or logical message not in a group) when GMLOGO is not specified. For example, if the last message in the group happens to precede the first message in the group on the queue, using GMBRWN to browse beyond the end of the group, specifying MOSEQN with *MDSEQ* set to 1 (to find the first message of the next group) would return again the first message in the group already browsed. This could happen immediately, or a number of MQGET calls later (if there are intervening groups).

The possibility of an infinite loop can be avoided by opening the queue twice for browse:

- Use the first handle to browse only the first message in each group.
- Use the second handle to browse only the messages within a specific group.
- Use the MO* options to move the second browse cursor to the position of the first browse cursor, before browsing the messages in the group.
- Do not use GMBRWN to browse beyond the end of a group.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue, is separate from the group and segment information that it retains for MQGET calls that remove messages from the queue.

This option is not valid with any of the following options:

- GMBRWF
- GMBRWC
- GMMUC
- GMSYP
- GMPSYP
- GMUNLK

It is also an error if the queue was not opened for browse.

GMBRWC

Browse message under browse cursor.

This option causes the message pointed to by the browse cursor to be retrieved nondestructively, regardless of the MO* options specified in the *GMMO* field in MQGMO.

The message pointed to by the browse cursor is the one that was last retrieved using either the GMBRWF or the GMBRWN option. The call fails if neither of these calls has been issued for this queue since it was opened, or if the message that was under the browse cursor has since been retrieved destructively.

The position of the browse cursor is not changed by this call.

The GMMUC option can then be used with a nonbrowse MQGET call if required, to remove the message from the queue.

The browse cursor is not moved by a nonbrowse MQGET call using the same *HOBJ* handle. Nor is it moved by a browse MQGET call that returns a completion code of CCFAIL, or a reason code of RC2080.

If GMBRWC is specified with GMLK:

- If there is already a message locked, it must be the one under the cursor, so that is returned without unlocking and relocking it; the message remains locked.
- If there is no locked message, the message under the browse cursor (if there is one) is locked and returned to the application; if there is no message under the browse cursor the call fails.

If GMBRWC is specified without GMLK:

- If there is already a message locked, it must be the one under the cursor. This message is returned to the application and then unlocked. Because the message is now unlocked, there is no guarantee that it can be browsed again, or retrieved destructively (it might be retrieved destructively by another application getting messages from the queue).
- If there is no locked message, the message under the browse cursor (if there is one) is returned to the application; if there is no message under the browse cursor the call fails.

If GMCMPM is specified with GMBRWC, the browse cursor must identify a message with a *MDOFF* field in MQMD that is zero. If this condition is not satisfied, the call fails with reason code RC2246 .

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue, is separate from the group and segment information that it retains for MQGET calls that remove messages from the queue.

This option is not valid with any of the following options:

- GMBRWF
- GMBRWN
- GMMUC
- GMSYP
- GMPSYP
- GMUNLK

It is also an error if the queue was not opened for browse.

GMMUC

Get message under browse cursor.

This option causes the message pointed to by the browse cursor to be retrieved, regardless of the MO* options specified in the *GMMO* field in MQGMO. The message is removed from the queue.

The message pointed to by the browse cursor is the one that was last retrieved using either the GMBRWF or the GMBRWN option.

If GMCMPM is specified with GMMUC, the browse cursor must identify a message with a *MDOFF* field in MQMD that is zero. If this condition is not satisfied, the call fails with reason code RC2246 .

This option is not valid with any of the following options:

- GMBRWF
- GMBRWC
- GMBRWN
- GMUNLK

It is also an error if the queue was not opened both for browse and for input. If the browse cursor is not currently pointing to a retrievable message, an error is returned by the MQGET call.

Lock options: The following options relate to locking messages on the queue:

GMLK

Lock message.

This option locks the message that is browsed, so that the message becomes invisible to any other handle open for the queue. The option can be specified only if one of the following options is also specified:

- GMBRWF
- GMBRWN
- GMBRWC

Only one message can be locked per queue handle, but this can be a logical message or a physical message:

- If GMCMPM is specified, all the message segments that make up the logical message are locked to the queue handle (if they are all present on the queue and available for retrieval).
- If GMCMPM is not specified, only a single physical message is locked to the queue handle. If this message happens to be a segment of a logical message, the locked segment prevents other applications using GMCMPM to retrieve or browse the logical message.

The locked message is always the one under the browse cursor, and the message can be removed from the queue by a later MQGET call that specifies the GMMUC option. Other MQGET calls using the queue handle can also remove the message (for example, a call that specifies the message identifier of the locked message).

If the call returns completion code CCFAIL, or CCWARN with reason code RC2080, no message is locked.

If the application decides not to remove the message from the queue, the lock is released by:

- Issuing another MQGET call for this handle, with either GMBRWF or GMBRWN specified (with or without GMLK); the message is unlocked if the call completes with CCOK or CCWARN, but remains locked if the call completes with CCFAIL. However, the following exceptions apply:
 - The message is not unlocked if CCWARN is returned with RC2080.
 - The message is unlocked if CCFAIL is returned with RC2033.

If GMLK is also specified, the message returned is locked. If GMLK is not specified, there is no locked message after the call.

If GMWT is specified, and no message is immediately available, the unlock on the original message occurs before the start of the wait (providing the call is otherwise free from error).

- Issuing another MQGET call for this handle, with GMBRWC (without GMLK); the message is unlocked if the call completes with CCOK or CCWARN, but remains locked if the call completes with CCFAIL. However, the following exception applies:
 - The message is not unlocked if CCWARN is returned with RC2080.
- Issuing another MQGET call for this handle with GMUNLK.
- Issuing an MQCLOSE call for this handle (either explicitly, or implicitly by the application ending).

No special open option is required to specify this option, other than OOBROW, which is needed in order to specify the accompanying browse option.

This option is not valid with any of the following options:

- GMSYP
- GMPSYP
- GMUNLK

GMUNLK

Unlock message.

The message to be unlocked must have been previously locked by an MQGET call with the GMLK option. If there is no message locked for this handle, the call completes with CCWARN and RC2209 .

The *MSGDSC*, *BUFLN*, *BUFFER*, and *DATLEN* parameters are not checked or altered if GMUNLK is specified. No message is returned in *BUFFER*.

No special open option is required to specify this option (although OOBROW is needed to issue the lock request in the first place).

This option is not valid with any options except the following:

- GMNWT
- GMNSYP

Both of these options are assumed whether specified or not.

Message-data options: The following options relate to the processing of the message data when the message is read from the queue:

GMATM

Allow truncation of message data.

If the message buffer is too small to hold the complete message, this option allows the MQGET call to fill the buffer with as much of the message as the buffer can hold, issue a warning completion code, and complete its processing. This means:

- When browsing messages, the browse cursor is advanced to the returned message.
- When removing messages, the returned message is removed from the queue.
- Reason code RC2079 is returned if no other error occurs.

Without this option, the buffer is still filled with as much of the message as it can hold, a warning completion code is issued, but processing is not completed. This means:

- When browsing messages, the browse cursor is not advanced.
- When removing messages, the message is not removed from the queue.
- Reason code RC2080 is returned if no other error occurs.

GMCONV

Convert message data.

This option requests that the application data in the message is converted, to conform to the *MDCSI* and *MDENC* values specified in the *MSGDSC* parameter on the MQGET call, before the data is copied to the *BUFFER* parameter.

The *MDFMT* field specified when the message was put is assumed by the conversion process to identify the nature of the data in the message. Conversion of the message data is by the queue manager for built-in formats, and by a user-written exit for other formats.

- If conversion is performed successfully, the *MDCSI* and *MDENC* fields specified in the *MSGDSC* parameter are unchanged on return from the MQGET call.
- If conversion cannot be performed successfully (but the MQGET call otherwise completes without error), the message data is returned unconverted, and the *MDCSI* and *MDENC* fields in *MSGDSC* are set to the values for the unconverted message. The completion code is CCWARN in this case.

In either case, therefore, these fields describe the character-set identifier and encoding of the message data that is returned in the *BUFFER* parameter.

See the *MDFMT* field described in [“MQMD - Message descriptor” on page 1068](#) for a list of format names for which the queue manager performs the conversion.

Group and segment options: The following options relate to the processing of messages in groups and segments of logical messages. These definitions might be of help in understanding the options:

Physical message

This is the smallest unit of information that can be placed on or removed from a queue; it often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor (MQMD). Generally, physical messages are distinguished by differing values for the message identifier (*MDMID* field in MQMD), although this is not enforced by the queue manager.

Logical message

This is a single unit of application information. In the absence of system constraints, a logical message would be the same as a physical message. But where logical messages are large, system constraints might make it advisable or necessary to split a logical message into two or more physical messages, called segments.

A logical message that has been segmented consists of two or more physical messages that have the same nonnull group identifier (*MDGID* field in MQMD), and the same message sequence number (*MDSEQ* field in MQMD). The segments are distinguished by differing values for the segment offset (*MDOFF* field in MQMD), which gives the offset of the data in the physical message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message typically have differing message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a nonnull group identifier, although in this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages for which segmentation has been inhibited by the sending application have a null group identifier (GINONE), unless the logical message belongs to a message group.

Message group

This is a set of one or more logical messages that have the same nonnull group identifier. The logical messages in the group are distinguished by differing values for the message sequence number, which is an integer in the range 1 through n, where n is the number of logical messages in the group. If one or more of the logical messages is segmented, there are more than n physical messages in the group.

GMLOGO

Messages in groups and segments of logical messages are returned in logical order.

This option controls the order in which messages are returned by successive MQGET calls for the queue handle. The option must be specified on each of those calls in order to have an effect.

If GMLOGO is specified for successive MQGET calls for the queue handle, messages in groups are returned in the order given by their message sequence numbers, and segments of logical messages are returned in the order given by their segment offsets. This order might be different from the order in which those messages and segments occur on the queue.

Note: Specifying GMLOGO has no adverse consequences on messages that do not belong to groups and that are not segments. In effect, such messages are treated as though each belonged to a message group consisting of only one message. Thus it is perfectly safe to specify GMLOGO when retrieving messages from queues that might contain a mixture of messages in groups, message segments, and unsegmented messages not in groups.

To return the messages in the required order, the queue manager retains the group and segment information between successive MQGET calls. This information identifies the current message group and current logical message for the queue handle, the current position within the group and logical message, and whether the messages are being retrieved within a unit of work. Because the queue manager retains this information, the application does not need to set the group and segment information before each MQGET call. Specifically, it means that the application does not need to set the *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD. However, the application does need to set the GMSYP or GMNSYP option correctly on each call.

When the queue is opened, there is no current message group and no current logical message. A message group becomes the current message group when a message that has the MFMIG flag is

returned by the MQGET call. With GMLOGO specified on successive calls, that group remains the current group until a message is returned that has:

- MFLMIG without MFSEG (that is, the last logical message in the group is not segmented), or
- MFLMIG with MFLSEG (that is, the message returned is the last segment of the last logical message in the group).

When such a message is returned, the message group is terminated, and on successful completion of that MQGET call there is no longer a current group. In a similar way, a logical message becomes the current logical message when a message that has the MFSEG flag is returned by the MQGET call, and that logical message is terminated when the message that has the MFLSEG flag is returned.

If no selection criteria are specified, successive MQGET calls return (in the correct order) the messages for the first message group on the queue, then the messages for the second message group, and so on, until there are no more messages available. It is possible to select the particular message groups returned by specifying one or more of the following options in the *GMMO* field:

- MOMSGI
- MOCORI
- MOGRPI

However, these options are effective only when there is no current message group or logical message; see the *GMMO* field described in this topic.

Table 593 on page 1047 shows the values of the *MDMID*, *MDCID*, *MDGID*, *MDSEQ*, and *MDOFF* fields that the queue manager looks for when attempting to find a message to return on the MQGET call. This applies both to removing messages from the queue, and browsing messages on the queue. The columns in the table have the following meanings:

LOG ORD

Indicates whether the GMLOGO option is specified on the call.

Cur grp

Indicates whether a current message group exists before the call.

Cur log msg

Indicates whether a current logical message exists before the call.

Other columns

Show the values that the queue manager looks for. "Previous" denotes the value returned for the field in the previous message for the queue handle.

Table 593. MQGET options relating to messages in groups and segments of logical messages							
Options you specify	Group and log-msg status before call		Values the queue manager looks for				
LOG ORD	Cur grp	Cur log msg	<i>MDMID</i>	<i>MDCID</i>	<i>MDGID</i>	<i>MDSEQ</i>	<i>MDOFF</i>
Yes	No	No	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	1	0
Yes	No	Yes	Any message identifier	Any correlation identifier	Previous group identifier	1	Previous offset + previous segment length
Yes	Yes	No	Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number + 1	0

Table 593. MQGET options relating to messages in groups and segments of logical messages (continued)							
Options you specify	Group and log-msg status before call		Values the queue manager looks for				
Yes	Yes	Yes	Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number	Previous offset + previous segment length
No	Either	Either	Controlled by GMMO	Controlled by GMMO	Controlled by GMMO	Controlled by GMMO	Controlled by GMMO

When multiple message groups are present on the queue and eligible for return, the groups are returned in the order determined by the position on the queue of the first segment of the first logical message in each group (that is, the physical messages that have message sequence numbers of 1, and offsets of 0, determine the order in which eligible groups are returned).

The GMLOGO option affects units of work as follows:

- If the first logical message or segment in a group is retrieved within a unit of work, all the other logical messages and segments in the group must be retrieved within a unit of work, if the same queue handle is used. However, they need not be retrieved within the same unit of work. This allows a message group consisting of many physical messages to be split across two or more consecutive units of work for the queue handle.
- If the first logical message or segment in a group is *not* retrieved within a unit of work, none of the other logical messages and segments in the group can be retrieved within a unit of work, if the same queue handle is used.

If these conditions are not satisfied, the MQGET call fails with reason code RC2245 .

When GMLOGO is specified, the MQGMO supplied on the MQGET call must not be less than GMVER2, and the MQMD must not be less than MDVER2. If this condition is not satisfied, the call fails with reason code RC2256 or RC2257 , as appropriate.

If GMLOGO is not specified for successive MQGET calls for the queue handle, messages are returned without regard for whether they belong to message groups, or whether they are segments of logical messages. This means that messages or segments from a particular group or logical message might be returned out of order, or they might be intermingled with messages or segments from other groups or logical messages, or with messages that are not in groups and are not segments. In this situation, the particular messages that are returned by successive MQGET calls is controlled by the MO* options specified on those calls (see the GMMO field described in [“MQGMO - Get-message options” on page 1035](#) for details of these options).

This is the technique that can be used to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the MDGID, MDSEQ, MDOFF, and GMMO fields to the appropriate values, and then issue the MQGET call with GMSYP or GMNSYP set as needed, but without specifying GMLOGO. If this call is successful, the queue manager retains the group and segment information, and subsequent MQGET calls using that queue handle can specify GMLOGO as normal.

The group and segment information that the queue manager retains for the MQGET call is separate from the group and segment information that it retains for the MQPUT call. In addition, the queue manager retains separate information for:

- MQGET calls that remove messages from the queue.
- MQGET calls that browse messages on the queue.

For any given queue handle, the application is free to mix MQGET calls that specify GMLOGO with MQGET calls that do not, but the following points must be noted:

- If GMLOGO is not specified, each successful MQGET call causes the queue manager to set the saved group and segment information to the values corresponding to the message returned; this replaces the existing group and segment information retained by the queue manager for the queue handle. Only the information appropriate to the action of the call (browse or remove) is modified.
- If GMLOGO is not specified, the call does not fail if there is a current message group or logical message; the call might however succeed with a CCWARN completion code. [Table 594 on page 1049](#) shows the various cases that can arise. In these cases, if the completion code is not CCOK, the reason code is one of the following:
 - RC2241
 - RC2242
 - RC2245

Note: The queue manager does not check the group and segment information when browsing a queue, or when closing a queue that was opened for browse but not input; in those cases the completion code is always CCOK (assuming no other errors).

<i>Table 594. Outcome when MQGET or MQCLOSE call is not consistent with group and segment information</i>		
Current call is	Previous call was MQGET with GMLOGO	Previous call was MQGET without GMLOGO
MQGET with GMLOGO	CCFAIL	CCFAIL
MQGET without GMLOGO	CCWARN	CCOK
MQCLOSE with an unterminated group or logical message	CCWARN	CCOK

Applications that simply want to retrieve messages and segments in logical order are recommended to specify GMLOGO, as this is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications might need more control than provided by the GMLOGO option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *MDMID*, *MDCID*, *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD, and the MO* options in GMMO in MQGMO, are set correctly, before each MQGET call.

For example, an application that wants to forward physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, should not specify GMLOGO. This is because in a complex network with multiple paths between sending and receiving queue managers, the physical messages might arrive out of order. By not specifying GMLOGO and the corresponding PMLOGO on the MQPUT call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without having to wait for the next one in logical order to arrive.

GMLOGO can be specified with any of the other GM* options, and with various of the MO* options in appropriate circumstances.

GMCMPM

Only complete logical messages are retrievable.

This option specifies that only a complete logical message can be returned by the MQGET call. If the logical message is segmented, the queue manager reassembles the segments and returns the complete logical message to the application; the fact that the logical message was segmented is not apparent to the application retrieving it.

Note: This is the only option that causes the queue manager to reassemble message segments. If not specified, segments are returned individually to the application if they are present on the

queue (and they satisfy the other selection criteria specified on the MQGET call). Applications that do not want to receive individual segments should therefore always specify GMCMPM.

To use this option, the application must provide a buffer which is large enough to accommodate the complete message, or specify the GMATM option.

If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying GMCMPM prevents the retrieval of segments belonging to incomplete logical messages. However, those message segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there might be no retrievable logical messages, even though *CurrentQDepth* is greater than zero.

For persistent messages, the queue manager can reassemble the segments only within a unit of work:

- If the MQGET call is operating within a user-defined unit of work, that unit of work is used. If the call fails part way through the reassembly process, the queue manager reinstates on the queue any segments that were removed during reassembly. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work just for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically (the application does not need to do this). If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work does exist, the queue manager is unable to perform reassembly. If the message does not require reassembly, the call can still succeed. But if the message does require reassembly, the call fails with reason code RC2255 .

For nonpersistent messages, the queue manager does not require a unit of work to be available in order to perform reassembly.

Each physical message that is a segment has its own message descriptor. For the segments constituting a single logical message, most of the fields in the message descriptor is the same for all segments in the logical message - typically it is only the *MDMID*, *MDOFF*, and *MDMFL* fields that differ between segments in the logical message. However, if a segment is placed on a dead-letter queue at an intermediate queue manager, the DLQ handler retrieves the message specifying the GMCONV option, and this might result in the character set or encoding of the segment being changed. If the DLQ handler successfully sends the segment on its way, the segment might have a character set or encoding that differs from the other segments in the logical message when the segment finally arrives at the destination queue manager.

A logical message consisting of segments in which the *MDCSI*, *MDENC*, or both fields differ cannot be reassembled by the queue manager into a single logical message. Instead, the queue manager reassembles and returns the first few consecutive segments at the start of the logical message that have the same character-set identifiers and encodings, and the MQGET call completes with completion code CCWARN and reason code RC2243 or RC2244 , as appropriate. This happens regardless of whether GMCONV is specified. To retrieve the remaining segments, the application must reissue the MQGET call without the GMCMPM option, retrieving the segments one by one. GMLOGO can be used to retrieve the remaining segments in order.

It is also possible for an application which puts segments to set other fields in the message descriptor to values that differ between segments. However, there is no advantage in doing this if the receiving application uses GMCMPM to retrieve the logical message. When the queue manager reassembles a logical message, it returns in the message descriptor the values from the message descriptor for the first segment; the only exception is the *MDMFL* field, which the queue manager sets to indicate that the reassembled message is the only segment.

If GMCMPM is specified for a report message, the queue manager performs special processing. The queue manager checks the queue to see if all the report messages of that report type relating

to the different segments in the logical message are present on the queue. If they are, they can be retrieved as a single message by specifying GMCMPM. For this to be possible, either the report messages must be generated by a queue manager or MCA which supports segmentation, or the originating application must request at least 100 bytes of message data (that is, the appropriate RO*D or RO*F options must be specified). If less than the full amount of application data is present for a segment, the missing bytes are replaced by nulls in the report message returned.

If GMCMPM is specified with GMMUC or GMBRWC, the browse cursor must be positioned on a message with a *MDOFF* field in MQMD that has a value of 0. If this condition is not satisfied, the call fails with reason code RC2246 .

GMCMPM implies GMASGA, which need not therefore be specified.

GMCMPM can be specified with any of the other GM* options apart from GMPSTP, and with any of the MO* options apart from MOOFFS.

GMAMSA

All messages in group must be available.

This option specifies that messages in a group become available for retrieval only when all messages in the group are available. If the queue contains message groups with some of the messages missing (perhaps because they have been delayed in the network and have not yet arrived), specifying GMAMSA prevents retrieval of messages belonging to incomplete groups. However, those messages still contribute to the value of the *CurrentQDepth* queue attribute; this means that there might be no retrievable message groups, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code RC2033 is returned after the specified wait interval (if any) has expired.

The processing of GMAMSA depends on whether GMLOGO is also specified:

- If both options are specified, GMAMSA affects *only* when there is no current group or logical message. If there *is* a current group or logical message, GMAMSA is ignored. This means that GMAMSA can remain on when processing messages in logical order.
- If GMAMSA is specified without GMLOGO, GMAMSA always has an effect. This means that the option must be turned off after the first message in the group has been removed from the queue, in order to be able to remove the remaining messages in the group.

Successful completion of an MQGET call specifying GMAMSA means that at the time that the MQGET call was issued, all the messages in the group were on the queue. However, be aware that other applications are still able to remove messages from the group (the group is not locked to the application that retrieves the first message in the group).

If this option is not specified, messages belonging to groups can be retrieved even when the group is incomplete.

GMAMSA implies GMASGA, which need not therefore be specified.

GMAMSA can be specified with any of the other GM* options, and with any of the MO* options.

GMASGA

All segments in a logical message must be available.

This option specifies that segments in a logical message become available for retrieval only when all segments in the logical message are available. If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying GMASGA prevents retrieval of segments belonging to incomplete logical messages. However those segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there might be no retrievable logical messages, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code RC2033 is returned after the specified wait interval (if any) has expired.

The processing of GMASGA depends on whether GMLOGO is also specified:

- If both options are specified, GMASGA has an effect only when there is no current logical message. If there is a current logical message, GMASGA is ignored. This means that GMASGA can remain on when processing messages in logical order.
- If GMASGA is specified without GMLOGO, GMASGA always has an effect. This means that the option must be turned off after the first segment in the logical message has been removed from the queue, in order to be able to remove the remaining segments in the logical message.

If this option is not specified, message segments can be retrieved even when the logical message is incomplete.

While both GMCMPM and GMASGA require all segments to be available before any of them can be retrieved, the former returns the complete message, whereas the latter allows the segments to be retrieved one by one.

If GMASGA is specified for a report message, the queue manager performs special processing. The queue manager checks the queue to see if there is at least one report message for each of the segments that make up the complete logical message. If there is, the GMASGA condition is satisfied. However, the queue manager does not check the type of the report messages present, and so there might be a mixture of report types in the report messages relating to the segments of the logical message. As a result, the success of GMASGA does not imply that GMCMPM succeeds. If there is a mixture of report types present for the segments of a particular logical message, those report messages must be retrieved one by one.

GMASGA can be specified with any of the other GM* options, and with any of the MO* options.

Default option: If none of the options described are required, the following option can be used:

GMNONE

No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. GMNONE is defined to aid program documentation; it is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

The initial value of the *GMOPT* field is GMNWT.

GMRE1 (1 byte character string)

Reserved.

This is a reserved field. The initial value of this field is a blank character. This field is ignored if *GMVER* is less than GMVER2.

GMRL (10 digit signed integer)

Length of message data returned (bytes).

This is an output field that is set by the queue manager to the length in bytes of the message data returned by the MQGET call in the *BUFFER* parameter. If the queue manager does not support this capability, *GMRL* is set to the value RLUNDF.

When messages are converted between encodings or character sets, the message data can sometimes change size. On return from the MQGET call:

- If *GMRL* is not RLUNDF, the number of bytes of message data returned is given by *GMRL*.
- If *GMRL* has the value RLUNDF, the number of bytes of message data returned is typically given by the smaller of *BUFLN* and *DATLEN*, but can be less than this if the MQGET call completes with reason code RC2079. If this happens, the insignificant bytes in the *BUFFER* parameter are set to nulls.

The following special value is defined:

RLUNDF

Length of returned data not defined.

The initial value of this field is RLUNDF. This field is ignored if *GMVER* is less than GMVER3.

GMRQN (48 byte character string)

Resolved name of destination queue.

This is an output field which is set by the queue manager to the local name of the queue from which the message was retrieved, as defined to the local queue manager. This is different from the name used to open the queue if:

- An alias queue was opened (in which case, the name of the local queue to which the alias resolved is returned), or
- A model queue was opened (in which case, the name of the dynamic local queue is returned).

The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

GMRS2 (1 byte character string)

Reserved.

This is a reserved field. The initial value of this field is a blank character. This field is ignored if *GMVER* is less than GMVER4.

GMSEG (1 byte character string)

Flag indicating whether further segmentation is allowed for the message retrieved.

It has one of the following values:

SEGIHB

Segmentation not allowed.

SEGALW

Segmentation allowed.

This is an output field. The initial value of this field is SEGIHB. This field is ignored if *GMVER* is less than GMVER2.

GMSG1 (10 digit signed integer)

Signal.

This is a reserved field; its value is not significant. The initial value of this field is 0.

GMSG2 (10 digit signed integer)

Signal identifier.

This is a reserved field; its value is not significant.

GMSID (4 byte character string)

Structure identifier.

The value must be:

GMSIDV

Identifier for get-message options structure.

This field is always an input field. The initial value of this field is GMSIDV.

GMSST (1 byte character string)

Flag indicating whether message retrieved is a segment of a logical message.

It has one of the following values:

SSNSEG

Message is not a segment.

SSSEG

Message is a segment, but is not the last segment of the logical message.

SSLSEG

Message is the last segment of the logical message.

This is also the value returned if the logical message consists of only one segment.

This field is an output field. The initial value of this field is SSNSEG. This field is ignored if *GMVER* is less than GMVER2.

GMTOK (16 byte bit string)

Message token.

This is a reserved field; its value is not significant. The following special value is defined:

MTKNON

No message token.

The value is binary zero for the length of the field.

The length of this field is given by LNMTOk. The initial value of this field is MTKNON. This field is ignored if *GMVER* is less than GMVER3.

GMVER (10 digit signed integer)

Structure version number.

The value must be one of the following:

GMVER1

Version-1 get-message options structure.

GMVER2

Version-2 get-message options structure.

GMVER3

Version-3 get-message options structure.

GMVER4

Version-4 get-message options structure.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

GMVERC

Current version of get-message options structure.

This field is always an input field. The initial value of this field is GMVER1.

GMVER (10 digit signed integer)

Structure version number.

The value must be one of the following:

GMVER1

Version-1 get-message options structure.

GMVER2

Version-2 get-message options structure.

GMVER3

Version-3 get-message options structure.

GMVER4

Version-4 get-message options structure.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

GMVERC

Current version of get-message options structure.

This field is always an input field. The initial value of this field is GMVER1.

GMWI (10 digit signed integer)

Wait interval.

This is the approximate time, expressed in milliseconds, that the MQGET call waits for a suitable message to arrive (that is, a message satisfying the selection criteria specified in the *MSGDSC* parameter of the MQGET call; see the *MDMID* field described in “MQMD - Message descriptor” on page 1068 for more details). If no suitable message has arrived after this time has elapsed, the call completes with CCFAIL and reason code RC2033.

GMWI is used with the *GMWT* option. It is ignored if this option is not specified. If it is specified, *GMWI* must be greater than or equal to zero, or the following special value:

WIULIM

Unlimited wait interval.

The initial value of this field is 0.

Initial values

Table 595. Initial values of fields in MQGMO		
Field name	Name of constant	Value of constant
<i>GMSID</i>	GMSIDV	'GMO¬'
<i>GMVER</i>	GMVER1	1
<i>GMOPT</i>	GMNWT	0
<i>GMWI</i>	None	0
<i>GMSG1</i>	None	0
<i>GMSG2</i>	None	0
<i>GMRQN</i>	None	Blanks
<i>GMMO</i>	MOMSGI + MOCORI	3
<i>GMGST</i>	GSNIG	'¬'
<i>GMSST</i>	SSNSEG	'¬'
<i>GMSEG</i>	SEGIHB	'¬'
<i>GMRE1</i>	None	'¬'
<i>GMTOK</i>	MTKNON	Nulls
<i>GMRL</i>	RLUNDF	-1
<i>GMRS2</i>	None	'¬'
<i>GMMH</i>	HMNONE	0

Notes:

1. The symbol ¬ represents a single blank character.

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..
D*
D* MQGMO Structure
D*
D* Structure identifier
D  GMSID          1      4    INZ('GMO ')
D* Structure version number
```

```

D  GMVER                5          8I 0 INZ(1)
D* Options that control the action of MQGET
D  GMOPT                9          12I 0 INZ(0)
D* Wait interval
D  GMWI                 13         16I 0 INZ(0)
D* Signal
D  MSG1                 17         20I 0 INZ(0)
D* Signal identifier
D  MSG2                 21         24I 0 INZ(0)
D* Resolved name of destination queue
D  GMRQN                25         72    INZ
D* Options controlling selection criteria used for MQGET
D  GMMO                 73         76I 0 INZ(3)
D* Flag indicating whether message retrieved is in a group
D  GMGST                77         77    INZ(' ')
D* Flag indicating whether message retrieved is a segment of a
D* logical message
D  GMSST                78         78    INZ(' ')
D* Flag indicating whether further segmentation is allowed for the message
D* retrieved
D  GMSEG                79         79    INZ(' ')
D* Reserved
D  GMRE1                80         80    INZ
D* Message token
D  GMTOK                81         96    INZ(X'0000000000000000-
D                                     0000000000000000')
D* Length of message data returned (bytes)
D  GMRL                 97        100I 0 INZ(-1)
D* Reserved
D  GMRS2               101        104I 0 INZ(0)
D* Message handle
D  GMMH                105        112I 0 INZ(0)

```

MQIIH - IMS information header

The MQIIH structure describes the information that must be present at the start of a message sent to the IMS bridge through IBM MQ for z/OS.

Overview

Format name: FMIMS.

Character set and encoding: Special conditions apply to the character set and encoding used for the MQIIH structure and application message data:

- Applications that connect to the queue manager that owns the IMS bridge queue must provide an MQIIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQIIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQIIH structure that is in any of the supported character sets and encodings; conversion of the MQIIH is performed by the receiving message channel agent connected to the queue manager that owns the IMS bridge queue.

Note: There is one exception to this. If the queue manager that owns the IMS bridge queue is using CICS for distributed queuing, the MQIIH must be in the character set and encoding of the queue manager that owns the IMS bridge queue.

- The application message data following the MQIIH structure must be in the same character set and encoding as the MQIIH structure. The *IICSI* and *IIENC* fields in the MQIIH structure cannot be used to specify the character set and encoding of the application message data.

A data-conversion exit must be provided by the user to convert the application message data if the data is not one of the built-in formats supported by the queue manager.

- [“Authenticating passtickets for IMS bridge applications” on page 1057](#)
- [“Fields” on page 1057](#)
- [“Initial values” on page 1060](#)
- [“RPG declaration” on page 1060](#)

Authenticating passtickets for IMS bridge applications

It is now possible for IBM MQ administrators to specify the application name to be used for authenticating passtickets, for IMS bridge applications. To do this, the application name is specified as a new attribute PTKTAPPL for the STGCLASS object definition, as a 1 to 8 character alphanumeric string.

A blank value means that authentication occurs as with previous releases of IBM MQ, that is, no application name flows on the authentication request, and the MVSxxxx value is used instead.

A value of 1 - 8 alphanumeric characters must follow the rules for passticket application names as described in the RACF publications.

IBM MQ Administrators and RACF administrators must both agree on the valid application names to be used. The RACF administrator must create a profile in the PTKTDATA class giving READ access to the user IDs of all applications that are to be granted access. The IBM MQ administrator must create or alter the required STGCLASS definitions that specify the application name to be used for passticket authentication.

For related information, see the *Script (MQSC) Command Reference*.

Fields

The MQIIH structure contains the following fields; the fields are described in **alphabetical order**:

IIAUT (8-byte character string)

RACF password or passticket.

This is optional; if specified, it is used with the user ID in the MQMD security context to build a UTOKEN that is sent to IMS to provide a security context. If it is not specified, the user ID is used without verification. This depends on the setting of the RACF switches, which may require an authenticator to be present.

This is ignored if the first byte is blank or null. The following special value may be used:

IAUNON

No authentication.

The length of this field is given by LNAUTH. The initial value of this field is IAUNON.

IICMT (1-byte character string)

Commit mode.

See the *OTMA Reference* for more information about IMS commit modes. The value must be one of the following:

ICMCTS

Commit then send.

This mode implies double queuing of output, but shorter region occupancy times. Fast-path and conversational transactions cannot run with this mode.

ICMSTC

Send then commit.

The initial value of this field is ICMCTS.

IICSI (10-digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

IIENC (10-digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

IIFLG (10-digit signed integer)

Flags.

The value must be:

IINONE

No flags.

The initial value of this field is IINONE.

IIFMT (8-byte character string)

IBM MQ format name of data that follows MQIIH.

This specifies the IBM MQ format name of the data that follows the MQIIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

IILEN (10-digit signed integer)

Length of MQIIH structure.

The value must be:

IILEN1

Length of IMS information header structure.

The initial value of this field is IILEN1.

IILTO (8-byte character string)

Logical terminal override.

This is placed in the IO PCB field. It is optional; if it is not specified the TPIPE name is used. It is ignored if the first byte is blank, or null.

The length of this field is given by LNLTOV. The initial value of this field is 8 blank characters.

IIMMN (8-byte character string)

Message format services map name.

This is placed in the IO PCB field. It is optional. On input it represents the MID, on output it represents the MOD. It is ignored if the first byte is blank or null.

The length of this field is given by LNMFMN. The initial value of this field is 8 blank characters.

IIRFM (8-byte character string)

IBM MQ format name of reply message.

This is the IBM MQ format name of the reply message that will be sent in response to the current message. The rules for coding this are the same as those for the *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

IIRSV (1-byte character string)

Reserved.

This is a reserved field; it must be blank.

IISEC (1-byte character string)

Security scope.

This indicates the required IMS security processing. The following values are defined:

ISSCHK

Check security scope.

An ACEE is built in the control region, but not in the dependent region.

ISSFUL

Full security scope.

A cached ACEE is built in the control region and a non-cached ACEE is built in the dependent region. If you use ISSFUL, you must ensure that the user ID for which the ACEE is built has access to the resources used in the dependent region.

If ISSCHK and ISSFUL are not specified for this field, ISSCHK is assumed.

The initial value of this field is ISSCHK.

IISID (4-byte character string)

Structure identifier.

The value must be:

IISIDV

Identifier for IMS information header structure.

The initial value of this field is IISIDV.

IITID (16-byte bit string)

Transaction instance identifier.

This field is used by output messages from IMS so is ignored on first input. If *IITST* is set to ITSIC, this must be provided in the next input, and all subsequent inputs, to enable IMS to correlate the messages to the correct conversation. The following special value may be used:

ITINON

No transaction instance id.

The length of this field is given by LNTIID. The initial value of this field is ITINON.

IITST (1-byte character string)

Transaction state.

This indicates the IMS conversation state. This is ignored on first input because no conversation exists. On subsequent inputs it indicates whether a conversation is active or not. On output it is set by IMS. The value must be one of the following:

ITSIC

In conversation.

ITSNIC

Not in conversation.

ITSARC

Return transaction state data in architected form.

This value is used only with the IMS /DISPLAY TRAN command. It causes the transaction state data to be returned in the IMS architected form instead of character form. See [Writing IMS transaction programs through IBM MQ](#) for further details.

The initial value of this field is ITSNIC.

IIVER (10-digit signed integer)

Structure version number.

The value must be:

IIVER1

Version number for IMS information header structure.

The following constant specifies the version number of the current version:

IIVERC

Current version of IMS information header structure.

The initial value of this field is IIVER1.

Initial values

Table 596. Initial values of fields in MQIIH		
Field name	Name of constant	Value of constant
IISID	IISIDV	'IIH¬'
IIVER	IIVER1	1
IILEN	IILEN1	84
IIENC	None	0
IICSI	None	0
IIFMT	FMNONE	Blanks
IIFLG	IINONE	0
IILTO	None	Blanks
IIMMN	None	Blanks
IIRFM	FMNONE	Blanks
IIAUT	IAUNON	Blanks
IITID	ITINON	Nulls
IITST	ITSNIC	'¬'
IICMT	ICMCTS	'0'
IISEC	ISSCHK	'C'
IIRSV	None	'¬'

Notes:

1. The symbol ¬ represents a single blank character.

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D*
D* MQIIH Structure
D*
D* Structure identifier
D IISID          1      4      INZ('IIH ')
D* Structure version number
D IIVER          5      8I 0 INZ(1)
D* Length of MQIIH structure
D IILEN          9      12I 0 INZ(84)
D* Reserved
D IIENC          13     16I 0 INZ(0)
D* Reserved
D IICSI          17     20I 0 INZ(0)
D* MQ format name of data that followsMQIIH
D IIFMT          21     28      INZ('      ')
D* Flags
D IIFLG          29     32I 0 INZ(0)
D* Logical terminal override
D IILTO          33     40      INZ
D* Message format services map name
D IIMMN          41     48      INZ
D* MQ format name of reply message
```

D	IIRFM	49	56	INZ('')
D*	RACF password or passticket			
D	IIAUT	57	64	INZ('')
D*	Transaction instance identifier			
D	IITID	65	80	INZ(X'0000000000000000-0000000000000000')
D				
D*	Transaction state			
D	IITST	81	81	INZ('')
D*	Commit mode			
D	IICMT	82	82	INZ('0')
D*	Security scope			
D	IISEC	83	83	INZ('C')
D*	Reserved			
D	IIRSV	84	84	INZ

MQIMPO - Inquire message property options

The MQIMPO structure allows applications to specify options that control how properties of messages are inquired.

Overview

Purpose: The structure is an input parameter on the MQINQMP call.

Character set and encoding: Data in MQIMPO must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1061](#)
- [“Initial values” on page 1067](#)
- [“RPG declaration” on page 1067](#)

Fields

The MQIMPO structure contains the following fields; the fields are described in **alphabetical order**:

IPOPT (10-digit signed integer)

The following options control the action of MQINQMP. You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations). Combinations of options that are not valid are noted; all other combinations are valid.

Value data options: The following options relate to the processing of the value data when the property is retrieved from the message.

IPCVAL

This option requests that the value of the property be converted to conform to the *IPREQCSI* and *IPREQENC* values specified before the MQINQMP call returns the property value in the *Value* area.

- If conversion is successful, the *IPRETCSI* and *IPRETENC* fields are set to the same as *IPREQCSI* and *IPREQENC* on return from the MQINQMP call.
- If conversion fails, but the MQINQMP call otherwise completes without error, the property value is returned unconverted.

If the property is a string, the *IPRETCSI* and *IPRETENC* fields are set to the character set and encoding of the unconverted string.

The completion code is CCWARN in this case, with reason code RC2466. The property cursor is advanced to the returned property.

If the property value expands during conversion, and exceeds the size of the *Value* parameter, the value is returned unconverted, with completion code CCFAIL; the reason code is set to RC2469.

The *DataLength* parameter of the MQINQMP call returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

This option also requests that:

- If the property name contains a wildcard, and
- The *IPRETNAMECHRP* field is initialized with an address or offset for the returned name, then the returned name is converted to conform to the *IPREQCSI* and *IPREQENC* values.
- If conversion is successful, the *VSCCSID* field of *IPRETNAMECHRP* and the encoding of the returned name are set to the input value of *IPREQCSI* and *IPREQENC*.
- If conversion fails, but the MQINQMP call otherwise completes without error or warning, the returned name is unconverted. The completion code is CCWARN in this case, with reason code RC2492.

The property cursor is advanced to the returned property. RC2466 is returned if both the value and the name are not converted.

If the returned name expands during conversion, and exceeds the size of the *VSBuFSIZE* field of the *RequestedName*, the returned string is left unconverted, with completion code CCFAIL and the reason code is set to RC2465.

The *VSLength* field of the MQCHARV structure returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

IPCTYP

This option requests that the value of the property be converted from its current data type, into the data type specified on the *Type* parameter of the MQINQMP call.

- If conversion is successful, the *Type* parameter is unchanged on return of the MQINQMP call.
- If conversion fails, but the MQINQMP call otherwise completes without error, the call fails with reason RC2470. The property cursor is unchanged.

If the conversion of the data type causes the value to expand during conversion, and the converted value exceeds the size of the *Value* parameter, the value is returned unconverted, with completion code CCFAIL and the reason code is set to RC2469.

The *DataLength* parameter of the MQINQMP call returns the length that the property value would have converted to, in order to allow the application to determine the size of the buffer required to accommodate the converted property value. The property cursor is unchanged.

If the value of the *Type* parameter of the MQINQMP call is not valid, the call fails with reason RC2473.

If the requested data type conversion is not supported, the call fails with reason RC2470. The following data type conversions are supported:

Property data type	Supported target data types
TYPBOL	TYPSTR, TYPI8, TYPI16, TYPI32, TYPI64
TYPBST	TYPSTR
TYPI8	TYPSTR, TYPI16, TYPI32, TYPI64
TYPI16	TYPSTR, TYPI32, TYPI64
TYPI32	TYPSTR, TYPI64

Property data type	Supported target data types
TYPI64	TYPSTR
TYPF32	TYPSTR, TYPF64
TYPF64	TYPSTR
TYPSTR	TYPBOL, TYPI8, TYPI16, TYPI32, TYPI64, TYPF32, TYPF64
TYPNUL	None

The general rules governing the supported conversions are as follows:

- Numeric property values can be converted from one data type to another, provided that no data is lost during the conversion.

For example, the value of a property with data type TYPI32 can be converted into a value with data type TYPI64, but cannot be converted into a value with data type TYPI16.

- A property value of any data type can be converted into a string.
- A string property value can be converted to any other data type provided the string is formatted correctly for the conversion. If an application attempts to convert a string property value that is not formatted correctly, IBM MQ returns reason code RC2472.
- If an application attempts a conversion that is not supported, IBM MQ returns reason code RC2470.

The specific rules for converting a property value from one data type to another are as follows:

- When converting a TYPBOL property value to a string, the value TRUE is converted to the string "TRUE", and the value false is converted to the string "FALSE".
- When converting a TYPBOL property value to a numeric data type, the value TRUE is converted to one, and the value FALSE is converted to zero.
- When converting a string property value to a TYPBOL value, the string "TRUE" , or "1" , is converted to TRUE, and the string "FALSE", or "0", is converted to FALSE.

Note that the terms "TRUE" and "FALSE" are not case sensitive.

Any other string cannot be converted; IBM MQ returns reason code RC2472.

- When converting a string property value to a value with data type TYPI8, TYPI16, TYPI32 or TYPI64, the string must have the following format:

```
[blanks][sign]digits
```

The meanings of the components of the string are as follows:

blanks

Optional leading blank characters

sign

An optional plus sign (+) or minus sign (-) character.

digits

A contiguous sequence of digit characters (0-9). At least one digit character must be present.

After the sequence of digit characters, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

IBM MQ returns reason code RC2472 if the string is not formatted correctly.

- When converting a string property value to a value with data type TYPF32 or TYPF64, the string must have the following format:

```
[blanks][sign]digits[.digits][e_char[e_sign]e_digits]
```

The meanings of the components of the string are as follows:

blanks

Optional leading blank characters

sign

An optional plus sign (+) or minus sign (-) character.

digits

A contiguous sequence of digit characters (0-9). At least one digit character must be present.

e_char

An exponent character, which is either "E" or "e".

e_sign

An optional plus sign (+) or minus sign (-) character for the exponent.

e_digits

A contiguous sequence of digit characters (0-9) for the exponent. At least one digit character must be present if the string contains an exponent character.

After the sequence of digit characters, or the optional characters representing an exponent, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

IBM MQ returns reason code RC2472 if the string is not formatted correctly.

- When converting a numeric property value to a string, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".
- When converting a byte string property value to a string, each byte is converted to the two hexadecimal characters that represent the byte. For example, the byte array {0xF1, 0x12, 0x00, 0xFF} is converted to the string "F11200FF".

IPQLEN

Query the type and length of the property value. The length is returned in the *DataLength* parameter of the MQINQMP call. The property value is not returned.

If a *ReturnedName* buffer is specified, the *VSLength* field of the MQCHARV structure is filled in with the length of the property name. The property name is not returned.

Iteration options: The following options relate to iterating over properties, using a name with a wildcard character

IPINQF

Inquire on the first property that matches the specified name. After this call, a cursor is established on the property that is returned.

This is the default value.

The IPINQC option can subsequently be used with an MQINQMP call, if required, to inquire on the same property again.

Note that there is only one property cursor; therefore, if the property name, specified in the MQINQMP call, changes the cursor is reset.

This option is not valid with either of the following options:

IPINQN

IPINQC

IPINQN

Inquires on the next property that matches the specified name, continuing the search from the property cursor. The cursor is advanced to the property that is returned.

If this is the first MQINQMP call for the specified name, then the first property that matches the specified name is returned.

The IPINQC option can subsequently be used with an MQINQMP call if required, to inquire on the same property again.

If the property under the cursor has been deleted, MQINQMP returns the next matching property following the one that has been deleted.

If a property is added that matches the wildcard, while an iteration is in progress, the property might or might not be returned during the completion of the iteration. The property is returned once the iteration restarts using IPINQF.

A property matching the wildcard that was deleted, while the iteration was in progress, is not returned subsequent to its deletion.

This option is not valid with either of the following options:

IPINQF

IPINQC

IPINQC

Retrieve the value of the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired, using either the IPINQF or the IPINQN option.

The property cursor is reset when the message handle is reused, when the message handle is specified in the *MsgHandle* field of the MQGMO on an MQGET call, or when the message handle is specified in *OriginalMsgHandle* or *NewMsgHandle* fields of the MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established, or if the property pointed to by the property cursor has been deleted, the call fails with completion code CCFAIL and reason RC2471.

This option is not valid with either of the following options:

IPINQF

IPINQN

If none of the options previously described is required, the following option can be used:

IPNONE

Use this value to indicate that no other options have been specified; all options assume their default values.

IPNONE aids program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is always an input field. The initial value of this field is IPINQF.

IPREQCSI (10-digit signed integer)

The character set that the inquired property value is to be converted into if the value is a character string. This is also the character set into which the *ReturnedName* is to be converted when IPCVAL or IPCTYP is specified.

The initial value of this field is CSAPL.

IPREQENC (10-digit signed integer)

This is the encoding into which the inquired property value is to be converted when IPCVAL or IPCTYP is specified.

The initial value of this field is ENNAT.

IPRE1 (10-digit signed integer)

This is a reserved field. The initial value of this field is a blank character.

IPRETCSI (10-digit signed integer)

On output, this is the character set of the value returned if the *Type* parameter of the MQINQMP call is TYPSTR.

If the IPCVAL option is specified and conversion was successful, the *ReturnedCCSID* field, on return, is the same value as the value passed in.

The initial value of this field is zero.

IPRETENC (10-digit signed integer)

On output, this is the encoding of the value returned.

If the IPCVAL option is specified and conversion was successful, the *ReturnedEncoding* field, on return, is the same value as the value passed in.

The initial value of this field is ENNAT.

IPRETNAMCHRP (10-digit signed integer)

The actual name of the inquired property.

On input a string buffer can be passed in using the *VSPtr* or *VSOffset* field of the MQCHARV structure. The length of the string buffer is specified using the *VSBuFSIZE* field of the MQCHARV structure.

On return from the MQINQMP call, the string buffer is completed with the name of the property that was inquired, provided the string buffer was long enough to fully contain the name. The *VSLength* field of the MQCHARV structure is filled in with the length of the property name. The *VSCCSID* field of the MQCHARV structure is filled in to indicate the character set of the returned name, whether or not conversion of the name failed.

This is an input/output field. The initial value of this field is MQCHARV_DEFAULT.

IPSID (10-digit signed integer)

This is the structure identifier. The value must be:

IPSIDV

Identifier for inquire message property options structure.

This is always an input field. The initial value of this field is IPSIDV.

IPTYP (10-digit signed integer)

A string representation of the data type of the property.

If the property was specified in an MQRFH2 header and the MQRFH2 dt attribute is not recognized, this field can be used to determine the data type of the property. *TypeString* is returned in coded character set 1208 (UTF-8), and is the first eight bytes of the value of the dt attribute of the property that failed to be recognized

This is always an output field. The initial value of this field is the null string in the C programming language, and 8 blank characters in other programming languages.

IPVER (10-digit signed integer)

This is the structure version number. The value must be:

IPVER1

Version number for inquire message property options structure.

The following constant specifies the version number of the current version:

IPVERC

Current version of inquire message property options structure.

This is always an input field. The initial value of this field is IPVER1.

Initial values

Table 597. Initial values of fields in MQIPMO		
Field name	Name of constant	Value of constant
<i>IPSID</i>	IPSIDV	'IMPO'
<i>IPVER</i>	IPVER1	1
<i>IPOPT</i>	IPINQF	
<i>IPREQENC</i>	ENNAT	
<i>IPREQCSI</i>	CSAPL	
<i>IPRETENC</i>	ENNAT	
<i>IPRETCSI</i>	0	
<i>IPRE1</i>	0	
<i>IPRETAMCHRP</i>		
<i>IPTYP</i>		blanks

RPG declaration

```

D* MQIMPO Structure
D*
D*
D* Structure identifier
D IPSID      1   4 INZ('IMPO')
D*
D* Structure version number
D IPVER      5   8I 0 INZ(1)
D*
** Options that control the action of
D* MQINQMP
D IPOPT      9  12I 0 INZ(0)
D*
D* Requested encoding of Value
D IPREQENC   13  16I 0 INZ(273)
D*
** Requested character set identifier
D* of Value
D IPREQCSI   17  20I 0 INZ(-3)
D*
D* Returned encoding of Value
D IPRETENC   21  24I 0 INZ(273)
D*
** Returned character set identifier of
D* Value
D IPRETCSI   25  28I 0 INZ(0)
D*
D* Reserved
D IPRE1      29  32I 0 INZ(0)
D*
D* Returned property name
D* Address of variable length string
D IPRETAMCHRP 33  48* INZ(*NULL)
D* Offset of variable length string
D IPRETAMCHRO 49  52I 0 INZ(0)

```

```

D* Size of buffer
D IPRETNAMVSBS      53      56I 0 INZ(-1)
D* Length of variable length string
D IPRETNAMCHRL      57      60I 0 INZ(0)
D* CCSID of variable length string
D IPRETNAMCHRC      61      64I 0 INZ(-3)
D*
D* Property data type as a string
D IPTYP              65      72  INZ

```

MQMD - Message descriptor

Overview

Purpose: The MQMD structure contains the control information that accompanies the application data when a message travels between the sending and receiving applications. The structure is an input/output parameter on the MQGET, MQPUT, and MQPUT1 calls.

Version: The current version of MQMD is MDVER2. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQMD that is supported by the environment, but with the initial value of the *MDVER* field set to MDVER1. To use fields that are not present in the version-1 structure, the application must set the *MDVER* field to the version number of the version required.

A declaration for the version-1 structure is available with the name MQMD1.

Character set and encoding: Data in MQMD must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ MQI client, the structure must be in the character set and encoding of the client.

If the sending and receiving queue managers use different character sets or encodings, the data in MQMD is converted automatically. It is not necessary for the application to convert the MQMD.

- [“Using different versions of MQMD” on page 1068](#)
- [“Message context” on page 1069](#)
- [“Message expiry” on page 1069](#)
- [“Fields” on page 1069](#)
- [“Initial values” on page 1109](#)
- [“RPG declaration” on page 1110](#)

Using different versions of MQMD

A version-2 MQMD is generally equivalent to using a version-1 MQMD and prefixing the message data with an MQMDE structure. However, if all of the fields in the MQMDE structure have their default values, the MQMDE can be omitted. A version-1 MQMD plus MQMDE are used as described later in this section.

- On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *MDFMT* field in MQMD to FMMDE to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager assumes default values for the fields in the MQMDE.

Note: Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on the MQPUT and MQPUT1 calls. However, the queue manager does *not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

- On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a non-default value. The *MDFMT* field in MQMD will have the value FMMDE to indicate that an MQMDE is present.

The default values that the queue manager used for the fields in the MQMDE are the same as the initial values of those fields, shown in [Table 598 on page 1109](#).

When a message is on a transmission queue, some of the fields in MQMD are set to particular values; see [“MQXQH - Transmission-queue header” on page 1201](#) for details.

Message context

Certain fields in MQMD contain the message context. Typically:

- **Identity** context relates to the application that *originally* put the message
- **Origin** context relates to the application that *most recently* put the message
- **User** context relates to the application that *originally* put the message.

These two applications can be the same application, but they can also be different applications (for example, when a message is forwarded from one application to another).

Although identity and origin context usually have the meanings described previously, the content of both types of context fields in MQMD actually depends on the PM* options that are specified when the message is put. As a result, identity context does not necessarily relate to the application that originally put the message, and origin context does not necessarily relate to the application that most recently put the message - it depends on the design of the application suite.

There is one class of application that never alters message context, namely the message channel agent (MCA). MCAs that receive messages from remote queue managers use the context option PMSETA on the MQPUT or MQPUT1 call. This allows the receiving MCA to preserve exactly the message context that travelled with the message from the sending MCA. However, the result is that the origin context does not relate to the application that most recently put the message (the receiving MCA), but instead relates to an earlier application that put the message (possibly the originating application itself).

For more information see [Message context](#).

Message expiry

Messages that have expired on a loaded queue (a queue that has been opened) are automatically removed from the queue within a reasonable period of time after their expiry. Some other new features of this release of IBM MQ can lead to loaded queues being scanned less frequently than in the previous product version, however expired messages on loaded queues are always removed within a reasonable period of their expiry.

Fields

The MQMD structure contains the following fields; the fields are described in alphabetical order:

MDACC (32-byte bit string)

Accounting token.

This is part of the **identity context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

MDACC allows an application to cause work done as a result of the message to be appropriately charged. The queue manager treats this information as a string of bits and does not check its content.

When the queue manager generates this information, it is set as follows:

- The first byte of the field is set to the length of the accounting information present in the bytes that follow; this length is in the range zero through 30, and is stored in the first byte as a binary integer.
- The second and subsequent bytes (as specified by the length field) are set to the accounting information appropriate to the environment.
 - On z/OS the accounting information is set to:

- For z/OS batch, the accounting information from the JES JOB card or from a JES ACCT statement in the EXEC card (comma separators are changed to X'FF'). This information is truncated, if necessary, to 31 bytes.
- For TSO, the user's account number.
- For CICS, the LU 6.2 unit of work identifier (UEPUOWDS) (26 bytes).
- For IMS, the 8-character PSB name concatenated with the 16-character IMS recovery token.
- On IBM i, the accounting information is set to the accounting code for the job.
- On HP Integrity NonStop Server, and UNIX systems, the accounting information is set to the numeric user identifier, in ASCII characters.
- On Windows, the accounting information is set to a Windows NT security identifier (SID) in a compressed format. The SID uniquely identifies the user identifier stored in the *MDUID* field. When the SID is stored in the *MDACC* field, the 6-byte Identifier Authority (located in the third and subsequent bytes of the SID) is omitted. For example, if the Windows NT SID is 28 bytes long, 22 bytes of SID information are stored in the *MDACC* field.
- The last byte is set to the accounting-token type, one of the following values:

ATTCIC

CICS LUOW identifier.

ATTDOS

PC DOS default accounting token.

ATTWNT

Windows security identifier.

ATT400

IBM i accounting token.

ATTUNIX

UNIX systems numeric identifier.

ATTUSR

User-defined accounting token.

ATTUNK

Unknown accounting-token type.

The accounting-token type is set to an explicit value only in the following environments: AIX, HP-UX, IBM i, Solaris, Windows, plus IBM MQ MQI clients connected to these systems. In other environments, the accounting-token type is set to the value ATTUNK. In these environments the *MDPAT* field can be used to deduce the type of accounting token received.

- All other bytes are set to binary zero.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETI or PMSETA is specified in the *PMO* parameter. If neither PMSETI nor PMSETA is specified, this field is ignored on input and is an output-only field. For more information on message context, see [Message context](#) and [Controlling context information](#).

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDACC* that was transmitted with the message if it was put to a queue. This will be the value of *MDACC* that is kept with the message if it is retained (see description of PMRET in [“MQPMO - Put-message options”](#) on page 1131 for more details about retained publications) but is not used as the *MDACC* when the message is sent as a publication to subscribers since they provide a value to override *MDACC* in all publications sent to them. If the message has no context, the field is entirely binary zero.

This is an output field for the MQGET call.

This field is not subject to any translation based on the character set of the queue manager-the field is treated as a string of bits, and not as a string of characters.

The queue manager does nothing with the information in this field. The application must interpret the information if it wants to use the information for accounting purposes.

The following special value may be used for the *MDACC* field:

ACNONE

No accounting token is specified.

The value is binary zero for the length of the field.

The length of this field is given by LNAACC. The initial value of this field is ACNONE.

MDAID (32-byte character string)

Application data relating to identity.

This is part of the **identity context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

MDAID is information that is defined by the application suite, and can be used to provide additional information about the message or its originator. The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETI or PMSETA is specified in the *PMO* parameter. If a null character is present, the null and any following characters are converted to blanks by the queue manager. If neither PMSETI nor PMSETA is specified, this field is ignored on input and is an output-only field. For more information on message context, see [Message context](#) and [Controlling context information](#).

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDAID* that was transmitted with the message if it was put to a queue. This will be the value of *MDAID* that is kept with the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDAID* when the message is sent as a publication to subscribers since they provide a value to override *MDAID* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by LNAIDD. The initial value of this field is 32 blank characters.

MDAOD (4-byte character string)

Application data relating to origin.

This is part of the **origin context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

MDAOD is information that is defined by the application suite that can be used to provide additional information about the origin of the message. For example, it could be set by applications running with suitable user authority to indicate whether the identity data is trusted.

The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If PMSETA is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDAOD* that was transmitted with the message if it was put to a queue. This will be the value of *MDAOD* that is kept with the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDAOD* when the message is sent as a publication to subscribers since they provide a value to override *MDAOD* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by LNAORD. The initial value of this field is 4 blank characters.

MDBOC (10-digit signed integer)

Backout counter.

This is a count of the number of times the message has been previously returned by the MQGET call as part of a unit of work, and subsequently backed out. It is provided as an aid to the application in detecting processing errors that are based on message content. The count excludes MQGET calls that specified any of the GMBRW* options.

The accuracy of this count is affected by the *HardenGetBackout* queue attribute; see [“Attributes for queues”](#) on page 1324.

This is an output field for the MQGET call. It is ignored for the MQPUT and MQPUT1 calls. The initial value of this field is 0.

MDCID (24-byte bit string)

Correlation identifier.

This is a byte string that the application can use to relate one message to another, or to relate the message to other work that the application is performing. The correlation identifier is a permanent property of the message, and persists across restarts of the queue manager. Because the correlation identifier is a byte string and not a character string, the correlation identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 calls, the application can specify any value. The queue manager transmits this value with the message and delivers it to the application that issues the get request for the message.

If the application specifies PMNCID, the queue manager generates a unique correlation identifier which is sent with the message, and also returned to the sending application on output from the MQPUT or MQPUT1 call.

This generated correlation identifier is kept with the message if it is retained and is used as the correlation identifier when the message is sent as a publication to subscribers who specify CINONE in the *SDCID* field in the MQSD passed on the MQSUB call.

See [“MQPMO - Put-message options”](#) on page 1131 for more details about retained publications

When the queue manager or a message channel agent generates a report message, it sets the *MDCID* field in the way specified by the *MDREP* field of the original message, either ROCMTC or ROPCI. Applications which generate report messages should also do this.

For the MQGET call, *MDCID* is one of the five fields that can be used to select a particular message to be retrieved from the queue. See the description of the *MDMID* field for details of how to specify values for this field.

Specifying CINONE as the correlation identifier has the same effect as *not* specifying MOCORI, that is, *any* correlation identifier will match.

If the GMMUC option is specified in the *GMO* parameter on the MQGET call, this field is ignored.

On return from an MQGET call, the *MDCID* field is set to the correlation identifier of the message returned (if any).

The following special values may be used:

CINONE

No correlation identifier is specified.

The value is binary zero for the length of the field.

CINEWS

Message is the start of a new session.

This value is recognized by the CICS bridge as indicating the start of a new session, that is, the start of a new sequence of messages.

For the MQGET call, this is an input/output field. For the MQPUT and MQPUT1 calls, this is an input field if PMNCID is *not* specified, and an output field if PMNCID is specified. The length of this field is given by LNCID. The initial value of this field is CINONE.

MDCSI (10-digit signed integer)

This specifies the character set identifier of character data in the message.

Note: Character data in MQMD and the other IBM MQ data structures that are parameters on calls must be in the character set of the queue manager. This is defined by the queue manager's *CodedCharSetId* attribute; see [“Attributes for the queue manager” on page 1355](#) for details of this attribute.

The following special values can be used:

CSQM

Queue manager's character set identifier.

Character data in the message is in the queue manager's character set.

On the MQPUT and MQPUT1 calls, the queue manager changes this value in the MQMD sent with the message to the true character-set identifier of the queue manager. As a result, the value CSQM is never returned by the MQGET call.

CSINHT

Inherit character-set identifier of this structure.

Character data in the message is in the same character set as this structure; this is the queue manager's character set. (For MQMD only, CSINHT has the same meaning as CSQM).

The queue manager changes this value in the MQMD sent with the message to the actual character-set identifier of MQMD. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

CSEMBD

Embedded character set identifier.

Character data in the message is in a character set with the identifier that is contained within the message data itself. There can be any number of character-set identifiers embedded within the message data, applying to different parts of the data. This value must be used for PCF messages that contain data in a mixture of character sets. PCF messages have a format name of FMPCF.

Specify this value only on the MQPUT and MQPUT1 calls. If it is specified on the MQGET call, it prevents conversion of the message.

On the MQPUT and MQPUT1 calls, the queue manager changes the values CSQM and CSINHT in the MQMD sent with the message as described previously, but does not change the MQMD specified on the MQPUT or MQPUT1 call. No other check is carried out on the value specified.

Applications that retrieve messages should compare this field against the value the application is expecting; if the values differ, the application may need to convert character data in the message.

If the GMCONV option is specified on the MQGET call, this field is an input/output field. The value specified by the application is the coded character-set identifier to which the message data should be converted if necessary. If conversion is successful or unnecessary, the value is unchanged (except that the value CSQM or CSINHT is converted to the actual value). If conversion is unsuccessful, the value after the MQGET call represents the coded character-set identifier of the unconverted message that is returned to the application.

Otherwise, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is CSQM.

MDENC (10-digit signed integer)

Numeric encoding of message data.

This specifies the numeric encoding of numeric data in the message; it does not apply to numeric data in the MQMD structure itself. The numeric encoding defines the representation used for binary integers, packed-decimal integers, and floating-point numbers.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that the field is valid. The following special value is defined:

ENNAT

Native machine encoding.

The encoding is the default for the programming language and machine on which the application is running.

Note: The value of this constant depends on the programming language and environment. For this reason, applications must be compiled using the header, macro, COPY, or INCLUDE files appropriate to the environment in which the application will run.

Applications that put messages should normally specify ENNAT. Applications that retrieve messages should compare this field against the value ENNAT; if the values differ, the application may need to convert numeric data in the message. The GMCONV option can be used to request the queue manager to convert the message as part of the processing of the MQGET call.

If the GMCONV option is specified on the MQGET call, this field is an input/output field. The value specified by the application is the encoding to which the message data should be converted if necessary. If conversion is successful or unnecessary, the value is unchanged. If conversion is unsuccessful, the value after the MQGET call represents the encoding of the unconverted message that is returned to the application.

In other cases, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is ENNAT.

MDEXP (10-digit signed integer)

Message lifetime.

This is a period of time expressed in tenths of a second, set by the application that puts the message. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

The value is decremented to reflect the time the message spends on the destination queue, and also on any intermediate transmission queues if the put is to a remote queue. It may also be decremented by message channel agents to reflect transmission times, if these are significant. Likewise, an application forwarding this message to another queue might decrement the value if necessary, if it has retained the message for a significant time. However, the expiration time is treated as approximate, and the value need not be decremented to reflect small time intervals.

When the message is retrieved by an application using the MQGET call, the *MDEXP* field represents the amount of the original expiry time that still remains.

After a message's expiry time has elapsed, it becomes eligible to be discarded by the queue manager. In the current implementations, the message is discarded when a browse or nonbrowse MQGET call occurs that would have returned the message had it not already expired. For example, a nonbrowse MQGET call with the *GMMO* field in MQGMO set to MONONE reading from a FIFO ordered queue will cause all the expired messages to be discarded up to the first unexpired message. With a priority ordered queue, the same call will discard expired messages of higher priority and messages of an equal priority that arrived on the queue before the first unexpired message.

A message that has expired is never returned to an application (either by a browse or a non-browse MQGET call), so the value in the *MDEXP* field of the message descriptor after a successful MQGET call is either greater than zero, or the special value EIULIM.

If a message is put on a remote queue, the message may expire (and be discarded) while it is on an intermediate transmission queue, before the message reaches the destination queue.

A report is generated when an expired message is discarded, if the message specified one of the ROEXP* report options. If none of these options is specified, no such report is generated; the message

is assumed to be no longer relevant after this time period (perhaps because a later message has superseded it).

Any other program that discards messages based on expiry time must also send an appropriate report message if one was requested.

Note:

1. If a message is put with an *MDEXP* time of zero, the MQPUT or MQPUT1 call fails with reason code RC2013; no report message is generated in this case.
2. Since a message with an expiry time that has elapsed may not actually be discarded until later, there may be messages on a queue that have passed their expiry time, and which are not therefore eligible for retrieval. These messages nevertheless count towards the number of messages on the queue for all purposes, including depth triggering.
3. An expiration report is generated, if requested, when the message is actually discarded, not when it becomes eligible for discarding.
4. Discarding of an expired message, and the generation of an expiration report if requested, are never part of the application's unit of work, even if the message was scheduled for discarding as a result of an MQGET call operating within a unit of work.
5. If a nearly-expired message is retrieved by an MQGET call within a unit of work, and the unit of work is subsequently backed out, the message may become eligible to be discarded before it can be retrieved again.
6. If a nearly-expired message is locked by an MQGET call with GMLK, the message may become eligible to be discarded before it can be retrieved by an MQGET call with GMMUC; reason code RC2034 is returned on this subsequent MQGET call if that happens.
7. When a request message with an expiry time greater than zero is retrieved, the application can take one of the following actions when it sends the reply message:
 - Copy the remaining expiry time from the request message to the reply message.
 - Set the expiry time in the reply message to an explicit value greater than zero.
 - Set the expiry time in the reply message to EIULIM.

The action to take depends on the design of the application suite. However, the default action for putting messages to a dead-letter (undelivered-message) queue should be to preserve the remaining expiry time of the message, and to continue to decrement it.

8. Trigger messages are always generated with EIULIM.
9. A message (normally on a transmission queue) which has a *MDFMT* name of FMXQH has a second message descriptor within the MQXQH. It therefore has two *MDEXP* fields associated with it. The following additional points should be noted in this case:
 - When an application puts a message on a remote queue, the queue manager places the message initially on a local transmission queue, and prefixes the application message data with an MQXQH structure. The queue manager sets the values of the two *MDEXP* fields to be the same as that specified by the application.

If an application puts a message directly on a local transmission queue, the message data must already begin with an MQXQH structure, and the format name must be FMXQH (but the queue manager does not enforce this). In this case the application need not set the values of these two *MDEXP* fields to be the same. (The queue manager does not check that the *MDEXP* field within the MQXQH contains a valid value, or even that the message data is long enough to include it.)

 - When a message with a *MDFMT* name of FMXQH is retrieved from a queue (whether this is a normal or a transmission queue), the queue manager decrements *both* these *MDEXP* fields with the time spent waiting on the queue. No error is raised if the message data is not long enough to include the *MDEXP* field in the MQXQH.
 - The queue manager uses the *MDEXP* field in the separate message descriptor (that is, not the one in the message descriptor embedded within the MQXQH structure) to test whether the message is eligible for discarding.

- If the initial values of the two *MDEXP* fields were different, it is therefore possible for the *MDEXP* time in the separate message descriptor when the message is retrieved to be greater than zero (so the message is not eligible for discarding), while the time according to the *MDEXP* field in the MQXQH has elapsed. In this case the *MDEXP* field in the MQXQH is set to zero.

The following special value is recognized:

EIULIM

Unlimited lifetime.

The message has an unlimited expiration time.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is EIULIM.

MDFB (10-digit signed integer)

Feedback or reason code.

This is used with a message of type MTRPRT to indicate the nature of the report, and is only meaningful with that type of message. The field can contain one of the FB* values, or one of the RC* values. Feedback codes are grouped as follows:

FBNONE

No feedback provided.

FBSFST

Lowest value for system-generated feedback.

FBSLST

Highest value for system-generated feedback.

The range of system-generated feedback codes FBSFST through FBSLST includes the general feedback codes listed later in this section (FB*), and also the reason codes (RC*) that can occur when the message cannot be put on the destination queue.

FBAFST

Lowest value for application-generated feedback.

FBALST

Highest value for application-generated feedback.

Applications that generate report messages should not use feedback codes in the system range (other than FBQUIT), unless they want to simulate report messages generated by the queue manager or message channel agent.

On the MQPUT or MQPUT1 calls, the value specified must either be FBNONE, or be within the system range or application range. This is checked whatever the value of *MDMT*.

General feedback codes:

FBCOA

Confirmation of arrival on the destination queue (see ROCOA).

FBCOD

Confirmation of delivery to the receiving application (see ROCOD).

FBEXP

Message expired.

Message was discarded because it had not been removed from the destination queue before its expiry time had elapsed.

FBPAN

Positive action notification (see ROPAN).

FBNAN

Negative action notification (see RONAN).

FBQUIT

Application should end.

This can be used by a workload scheduling program to control the number of instances of an application program that are running. Sending an MTRPRT message with this feedback code to an instance of the application program indicates to that instance that it should stop processing. However, adherence to this convention is a matter for the application; it is not enforced by the queue manager.

IMS-bridge feedback codes: When the IMS bridge receives a nonzero IMS-OTMA sense code, the IMS bridge converts the sense code from hexadecimal to decimal, adds the value FBIERR (300), and places the result in the *MDFB* field of the reply message. This results in the feedback code having a value in the range FBIFST (301) through FBILST (399) when an IMS-OTMA error has occurred.

The following feedback codes can be generated by the IMS bridge:

FBDLZ

Data length zero.

A segment length was zero in the application data of the message.

FBDLN

Data length negative.

A segment length was negative in the application data of the message.

FBDLTB

Data length too big.

A segment length was too big in the application data of the message.

FBBUFO

Buffer overflow.

The value of one of the length fields would cause the data to overflow the message buffer.

FBLOB1

Length in error by one.

The value of one of the length fields was one byte too short.

FBIIH

MQIIH structure not valid or missing.

The *MDFMT* field in MQMD specifies FMIMS, but the message does not begin with a valid MQIIH structure.

FBNAFI

User ID not authorized for use in IMS.

The user ID contained in the message descriptor MQMD, or the password contained in the *IIAUT* field in the MQIIH structure, failed the validation performed by the IMS bridge. As a result the message was not passed to IMS.

FBIERR

Unexpected error returned by IMS.

An unexpected error was returned by IMS. Consult the IBM MQ error log on the system on which the IMS bridge resides for more information about the error.

FBIFST

Lowest value for IMS-generated feedback.

IMS-generated feedback codes occupy the range FBIFST (300) through FBILST (399). The IMS-OTMA sense code itself is *MDFB* minus FBIERR.

FBILST

Highest value for IMS-generated feedback.

CICS-bridge feedback codes: The following feedback codes can be generated by the CICS bridge:

FBCAAB

Application abended.

The application program specified in the message abended. This feedback code occurs only in the *DLREA* field of the MQDLH structure.

FBCANS

Application cannot be started.

The EXEC CICS LINK for the application program specified in the message failed. This feedback code occurs only in the *DLREA* field of the MQDLH structure.

FBCBRF

CICS bridge terminated abnormally without completing normal error processing.

FBCCSSE

Character set identifier not valid.

FBCIHE

CICS information header structure missing or not valid.

FBCCAE

Length of CICS commarea not valid.

FBCIE

Correlation identifier not valid.

FBCDLQ

Dead-letter queue not available.

The CICS bridge task was unable to copy a reply to this request to the dead-letter queue. The request was backed out.

FBCENE

Encoding not valid.

FBCINE

CICS bridge encountered an unexpected error.

This feedback code occurs only in the *DLREA* field of the MQDLH structure.

FBCNTA

User identifier not authorized or password not valid.

This feedback code occurs only in the *DLREA* field of the MQDLH structure.

FBCUBO

Unit of work backed out.

The unit of work was backed out, for one of the following reasons:

- A failure was detected while processing another request within the same unit of work.
- A CICS abend occurred while the unit of work was in progress.

FBCUWE

Unit-of-work control field *CIUOW* not valid.

MQ reason codes: For exception report messages, *MDFB* contains an MQ reason code. Among possible reason codes are:

RC2051

(2051, X'803') Put calls inhibited for the queue.

RC2053

(2053, X'805') Queue already contains maximum number of messages.

RC2035

(2035, X'7F3') Not authorized for access.

RC2056

(2056, X'808') No space available on disk for queue.

RC2048

(2048, X'800') Queue does not support persistent messages.

RC2031

(2031, X'7EF') Message length greater than maximum for queue manager.

RC2030

(2030, X'7EE') Message length greater than maximum for queue.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is FBNONE.

MDFMT (8-byte character string)

Format name of message data.

This is a name that the sender of the message may use to indicate to the receiver the nature of the data in the message. Any characters that are in the queue manager's character set may be specified for the name, but it is recommended that the name be restricted to the following:

- Uppercase A through Z
- Numeric digits 0 through 9

If other characters are used, it may not be possible to translate the name between the character sets of the sending and receiving queue managers.

The name should be padded with blanks to the length of the field, or a null character used to terminate the name before the end of the field; the null and any subsequent characters are treated as blanks. Do not specify a name with leading or embedded blanks. For the MQGET call, the queue manager returns the name padded with blanks to the length of the field.

The queue manager does not check that the name complies with the recommendations described previously.

Names beginning "MQ" in upper, lower, and mixed case have meanings that are defined by the queue manager; you should not use names beginning with these letters for your own formats. The queue manager built-in formats are:

FMNONE

No format name.

The nature of the data is undefined. This means that the data cannot be converted when the message is retrieved from a queue using the GMCONV option.

If GMCONV is specified on the MQGET call, and the character set or encoding of data in the message differs from that specified in the *MSGDSC* parameter, the message is returned with the following completion and reason codes (assuming no other errors):

- Completion code CCWARN and reason code RC2110 if the FMNONE data is at the beginning of the message.
- Completion code CCOK and reason code RCNONE if the FMNONE data is at the end of the message (that is, preceded by one or more MQ header structures). The MQ header structures are converted to the requested character set and encoding in this case.

FMADMN

Command server request/reply message.

The message is a command-server request or reply message in programmable command format (PCF). Messages of this format can be converted if the GMCONV option is specified on the MQGET call. For more information about using programmable command format messages, see [Using Programmable Command Formats](#).

FMCICS

CICS information header.

The message data begins with the CICS information header MQCIH, which is followed by the application data. The format name of the application data is given by the *CIFMT* field in the MQCIH structure.

FMCMD1

Type 1 command reply message.

The message is an MQSC command-server reply message containing the object count, completion code, and reason code. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMCMD2

Type 2 command reply message.

The message is an MQSC command-server reply message containing information about the object(s) requested. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMDLH

Dead-letter header.

The message data begins with the dead-letter header MQDLH. The data from the original message immediately follows the MQDLH structure. The format name of the original message data is given by the *DLFMT* field in the MQDLH structure; see [“MQDLH - Dead-letter header” on page 1023](#) for details of this structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

COA and COD reports are not generated for messages which have a *MDFMT* of FMDLH.

FMDH

Distribution-list header.

The message data begins with the distribution-list header MQDH; this includes the arrays of MQOR and MQPMR records. The distribution-list header may be followed by additional data. The format of the additional data (if any) is given by the *DHFMT* field in the MQDH structure; see [“MQDH - Distribution header” on page 1019](#) for details of this structure. Messages with format FMDH can be converted if the GMCONV option is specified on the MQGET call.

FMEVNT

Event message.

The message is an MQ event message that reports an event that occurred. Event messages have the same structure as programmable commands; for more information about this structure, see [Structures for commands and responses](#). For information about events, see [Event monitoring](#).

Version-1 event messages can be converted if the GMCONV option is specified on the MQGET call.

FMIMS

IMS information header.

The message data begins with the IMS information header MQIIH, which is followed by the application data. The format name of the application data is given by the *IIFMT* field in the MQIIH structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMIMVS

IMS variable string.

The message is an IMS variable string, which is a string of the form 11zzccc, where:

11

is a 2-byte length field specifying the total length of the IMS variable string item. This length is equal to the length of 11 (2 bytes), plus the length of zz (2 bytes), plus the length of the character string itself. 11 is a 2-byte binary integer in the encoding specified by the *MDENC* field.

zz

is a 2-byte field containing flags that are significant to IMS. *zz* is a byte string consisting of two 1-byte bit string fields, and is transmitted without change from sender to receiver (that is, *zz* is not subject to any conversion).

ccc

is a variable-length character string containing 11-4 characters. *ccc* is in the character set specified by the *MDCSI* field.

Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMMDE

Message-descriptor extension.

The message data begins with the message-descriptor extension *MQMDE*, and is optionally followed by other data (usually the application message data). The format name, character set, and encoding of the data which follows the *MQMDE* is given by the *MEFMT*, *MECSI*, and *MEENC* fields in the *MQMDE*. See [“MQMDE - Message descriptor extension” on page 1111](#) for details of this structure. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMPCF

User-defined message in programmable command format (PCF).

The message is a user-defined message that conforms to the structure of a programmable command format (PCF) message. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call. See [Using Programmable Command Formats](#) for more information about using programmable command format messages.

FMRMH

Reference message header.

The message data begins with the reference message header *MQRMH*, and is optionally followed by other data. The format name, character set, and encoding of the data is given by the *RMFMT*, *RMCSI*, and *RMENC* fields in the *MQRMH*. See [“MQRMH - Reference message header” on page 1157](#) for details of this structure. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMRFH

Rules and formatting header.

The message data begins with the rules and formatting header *MQRFH*, and is optionally followed by other data. The format name, character set, and encoding of the data (if any) is given by the *RFFMT*, *RFCSI*, and *RFENC* fields in the *MQRFH*. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMRFH2

Rules and formatting header version 2.

The message data begins with the version-2 rules and formatting header *MQRFH2*, and is optionally followed by other data. The format name, character set, and encoding of the optional data (if any) is given by the *RF2FMT*, *RF2CSI*, and *RF2ENC* fields in the *MQRFH2*. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMSTR

Message consisting entirely of characters.

The application message data can be either an SBCS string (single-byte character set), or a DBCS string (double-byte character set). Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMTM

Trigger message.

The message is a trigger message, described by the MQTM structure; see [“MQTM - Trigger message” on page 1191](#) for details of this structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMWIH

Work information header.

The message data begins with the work information header MQWIH, which is followed by the application data. The format name of the application data is given by the *WIFMT* field in the MQWIH structure.

FMXQH

Transmission queue header.

The message data begins with the transmission queue header MQXQH. The data from the original message immediately follows the MQXQH structure. The format name of the original message data is given by the *MDFMT* field in the MQMD structure which is part of the transmission queue header MQXQH. See [“MQXQH - Transmission-queue header” on page 1201](#) for details of this structure.

COA and COD reports are not generated for messages which have a *MDFMT* of FMXQH.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by LNFMT. The initial value of this field is FMNONE.

MDGID (24-byte bit string)

Group identifier.

This is a byte string that is used to identify the particular message group or logical message to which the physical message belongs. *MDGID* is also used if segmentation is allowed for the message. In all of these cases, *MDGID* has a non-null value, and one or more of the following flags is set in the *MDMFL* field:

- MFMIG
- MFLMIG
- MFSEG
- MFLSEG
- MFSEGA

If none of these flags is set, *MDGID* has the special null value GINONE.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, PMLOGO is specified.
- On the MQGET call, MOGRPI is *not* specified.

Consider using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *MDGID* is set to an appropriate value.

Message groups and segments can be processed correctly only if the group identifier is unique. For this reason, *applications should not generate their own group identifiers* ; instead, applications should do one of the following:

- If PMLOGO is specified, the queue manager automatically generates a unique group identifier for the first message in the group or segment of the logical message, and uses that group identifier for the remaining messages in the group or segments of the logical message, so the application does not need to take any special action. Consider using this procedure.
- If PMLOGO is *not* specified, the application should request the queue manager to generate the group identifier, by setting *MDGID* to GINONE on the first MQPUT or MQPUT1 call for a message in the group or segment of the logical message. The group identifier returned by the queue manager on output from that call should then be used for the remaining messages in the group or segments

of the logical message. If a message group contains segmented messages, the same group identifier must be used for all segments and messages in the group.

When PMLOGO is not specified, messages in groups and segments of logical messages can be put in any order (for example, in reverse order), but the group identifier must be allocated by the *first* MQPUT or MQPUT1 call that is issued for any of those messages.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in [PMOPT](#). On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message if the object opened is a single queue and not a distribution list, but leaves it unchanged if the object opened is a distribution list. In the latter case, if the application needs to know the group identifiers generated, the application must provide MQPMR records containing the *PRGID* field.

On input to the MQGET call, the queue manager uses the value detailed in [Table 1](#). On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The following special value is defined:

GINONE

No group identifier specified.

The value is binary zero for the length of the field. This is the value that is used for messages that are not in groups, not segments of logical messages, and for which segmentation is not allowed.

The length of this field is given by LNGID. The initial value of this field is GINONE. This field is ignored if MDVER is less than MDVER2.

MDMFL (10-digit signed integer)

Message flags.

These are flags that specify attributes of the message, or control its processing. The flags are divided into the following categories:

- Segmentation flag
- Status flags

These are described in turn.

Segmentation flags: When a message is too big for a queue, an attempt to put the message on the queue usually fails. Segmentation is a technique whereby the queue manager or application splits the message into smaller pieces called segments, and places each segment on the queue as a separate physical message. The application which retrieves the message can either retrieve the segments one by one, or request the queue manager to reassemble the segments into a single message which is returned by the MQGET call. The latter is achieved by specifying the GMCMPM option on the MQGET call, and supplying a buffer that is big enough to accommodate the complete message. (See [“MQGMO - Get-message options”](#) on page 1035 for details of the GMCMPM option.) Segmentation of a message can occur at the sending queue manager, at an intermediate queue manager, or at the destination queue manager.

You can specify one of the following to control the segmentation of a message:

MFSEGI

Segmentation inhibited.

This option prevents the message being broken into segments by the queue manager. If specified for a message that is already a segment, this option prevents the segment being broken into smaller segments.

The value of this flag is binary zero. This is the default.

MFSEGA

Segmentation allowed.

This option allows the message to be broken into segments by the queue manager. If specified for a message that is already a segment, this option allows the segment to be broken into smaller segments. MFSEGA can be set without either MFSEG or MFLSEG being set.

When the queue manager segments a message, the queue manager turns on the MFSEG flag in the copy of the MQMD that is sent with each segment, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call. For the last segment in the logical message, the queue manager also turns on the MFLSEG flag in the MQMD that is sent with the segment.

Note: Care is needed when messages are put with MFSEGA but without PMLOGO. If the message is:

- Not a segment, and
- Not in a group, and
- Not being forwarded,

the application must remember to reset the *MDGID* field to GINONE before *each* MQPUT or MQPUT1 call, in order to cause a unique group identifier to be generated by the queue manager for each message. If this is not done, unrelated messages could inadvertently end up with the same group identifier, which might lead to incorrect processing subsequently. See the descriptions of the *MDGID* field and the PMLOGO option for more information about when the *MDGID* field must be reset.

The queue manager splits messages into segments as necessary in order to ensure that the segments (plus any header data that may be required) fit on the queue. However, there is a lower limit for the size of a segment generated by the queue manager, and only the last segment created from a message can be smaller than this limit. (The lower limit for the size of an application-generated segment is one byte.) Segments generated by the queue manager may be of unequal length. The queue manager processes the message as follows:

- User-defined formats are split on boundaries which are multiples of 16 bytes. This means that the queue manager will not generate segments that are smaller than 16 bytes (other than the last segment).
- Built-in formats other than FMSTR are split at points appropriate to the nature of the data present. However, the queue manager never splits a message in the middle of an MQ header structure. This means that a segment containing a single MQ header structure cannot be split further by the queue manager, and as a result the minimum possible segment size for that message is greater than 16 bytes.

The second or later segment generated by the queue manager will begin with one of the following:

- An MQ header structure
- The start of the application message data
- Part-way through the application message data
- FMSTR is split without regard for the nature of the data present (SBCS, DBCS, or mixed SBCS/DBCS). When the string is DBCS or mixed SBCS/DBCS, this may result in segments which cannot be converted from one character set to another. The queue manager never splits FMSTR messages into segments that are smaller than 16 bytes (other than the last segment).
- The *MDFMT*, *MDCSI*, and *MDENC* fields in the MQMD of each segment are set by the queue manager to describe correctly the data present at the *start* of the segment; the format name will be either the name of a built-in format, or the name of a user-defined format.
- The *MDREP* field in the MQMD of segments with *MDOFF* greater than zero are modified as follows:
 - For each report type, if the report option is RO*D, but the segment cannot possibly contain any of the first 100 bytes of user data (that is, the data following any MQ header structures that may be present), the report option is changed to RO*.

The queue manager follows the previously rules, but otherwise splits messages unpredictably; do not make assumptions about where a message is split

For *persistent* messages, the queue manager can perform segmentation only within a unit of work:

- If the MQPUT or MQPUT1 call is operating within a user-defined unit of work, that unit of work is used. If the call fails partway through the segmentation process, the queue manager removes any segments that were placed on the queue as a result of the failing call. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work just for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically (the application does not need to do this). If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work *does* exist, the queue manager is unable to perform segmentation. If the message does not require segmentation, the call can still succeed. But if the message *does* require segmentation, the call fails with reason code RC2255.

For *nonpersistent* messages, the queue manager does not require a unit of work to be available in order to perform segmentation.

Special consideration must be given to data conversion of messages which may be segmented:

- If data conversion is performed only by the receiving application on the MQGET call, and the application specifies the GMCMPM option, the data-conversion exit will be passed the complete message for the exit to convert, and the fact that the message was segmented will not be apparent to the exit.
- If the receiving application retrieves one segment at a time, the data-conversion exit will be invoked to convert one segment at a time. The exit must therefore be capable of converting the data in a segment independently of the data in any of the other segments.

If the nature of the data in the message is such that arbitrary segmentation of the data on 16-byte boundaries may result in segments which cannot be converted by the exit, or the format is FMSTR and the character set is DBCS or mixed SBCS/DBCS, the sending application should itself create and put the segments, specifying MFSEGI to suppress further segmentation. In this way, the sending application can ensure that each segment contains sufficient information to allow the data-conversion exit to convert the segment successfully.

- If sender conversion is specified for a sending message channel agent (MCA), the MCA converts only messages which are not segments of logical messages; the MCA never attempts to convert messages which are segments.

This flag is an input flag on the MQPUT and MQPUT1 calls, and an output flag on the MQGET call. On the latter call, the queue manager also echoes the value of the flag to the *GMSEG* field in MQGMO.

The initial value of this flag is MFSEGI.

Status flags: These are flags that indicate whether the physical message belongs to a message group, is a segment of a logical message, both, or neither. One or more of the following can be specified on the MQPUT or MQPUT1 call, or returned by the MQGET call:

MFMI

Message is a member of a group.

MFLMI

Message is the last logical message in a group.

If this flag is set, the queue manager turns on MFMI in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

It is valid for a group to consist of only one logical message. If this is the case, MFLMI is set, but the *MDSEQ* field has the value one.

MFSEG

Message is a segment of a logical message.

When MFSEG is specified without MFLSEG, the length of the application message data in the segment (*excluding* the lengths of any MQ header structures that may be present) must be at least one. If the length is zero, the MQPUT or MQPUT1 call fails with reason code RC2253.

MFLSEG

Message is the last segment of a logical message.

If this flag is set, the queue manager turns on MFSEG in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

It is valid for a logical message to consist of only one segment. If this is the case, MFLSEG is set, but the *MDOFF* field has the value zero.

When MFLSEG is specified, it is permissible for the length of the application message data in the segment (*excluding* the lengths of any header structures that may be present) to be zero.

The application must ensure that these flags are set correctly when putting messages. If PMLOGO is specified, or was specified on the preceding MQPUT call for the queue handle, the settings of the flags must be consistent with the group and segment information retained by the queue manager for the queue handle. The following conditions apply to *successive* MQPUT calls for the queue handle when PMLOGO is specified:

- If there is no current group or logical message, all of these flags (and combinations of them) are valid.
- Once MFMIG has been specified, it must remain on until MFLMIG is specified. The call fails with reason code RC2241 if this condition is not satisfied.
- Once MFSEG has been specified, it must remain on until MFLSEG is specified. The call fails with reason code RC2242 if this condition is not satisfied.
- Once MFSEG has been specified without MFMIG, MFMIG must remain *off* until after MFLSEG has been specified. The call fails with reason code RC2242 if this condition is not satisfied.

Table 1 shows the valid combinations of the flags, and the values used for various fields.

These flags are input flags on the MQPUT and MQPUT1 calls, and output flags on the MQGET call. On the latter call, the queue manager also echoes the values of the flags to the *GMGST* and *GMSST* fields in MQGMO.

Default flags: The following can be specified to indicate that the message has default attributes:

MFNONE

No message flags (default message attributes).

This inhibits segmentation, and indicates that the message is not in a group and is not a segment of a logical message. MFNONE is defined to aid program documentation. It is not intended that this flag be used with any other, but as its value is zero, such use cannot be detected.

The *MDMFL* field is partitioned into subfields; for details see [“Report options and message flags” on page 1388](#).

The initial value of this field is MFNONE. This field is ignored if *MDVER* is less than MDVER2.

MDMID (24-byte bit string)

Message identifier.

This is a byte string that is used to distinguish one message from another. Generally, no two messages should have the same message identifier, although this is not disallowed by the queue manager. The message identifier is a permanent property of the message, and persists across restarts of the queue manager. Because the message identifier is a byte string and not a character string, the message identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 calls, if MINONE or PMNMID is specified by the application, the queue manager generates a unique message identifier ⁴ when the message is put, and places it in the message descriptor sent with the message. The queue manager also returns this message identifier in the message descriptor belonging to the sending application. The application can use this value to record information about particular messages, and to respond to queries from other parts of the application.

If the message is being put to a topic, the queue manager generates unique message identifiers as necessary for each message published. If PMNMID is specified by the application, the queue manager generates a unique message identifier to return on output. If MINONE is specified by the application, the value of the *MDMID* field in the MQMD is unchanged on return from the call.

See the description of PMRET in [PMOPT](#) for more details about retained publications.

If the message is being put to a distribution list, the queue manager generates unique message identifiers as necessary, but the value of the *MDMID* field in MQMD is unchanged on return from the call, even if MINONE or PMNMID was specified. If the application needs to know the message identifiers generated by the queue manager, the application must provide MQPMR records containing the *PRMID* field.

The sending application can also specify a particular value for the message identifier, other than MINONE; this stops the queue manager generating a unique message identifier. An application that is forwarding a message can use this facility to propagate the message identifier of the original message.

The queue manager does not itself make any use of this field except to:

- Generate a unique value if requested, as described previously
- Deliver the value to the application that issues the get request for the message
- Copy the value to the *MDCID* field of any report message that it generates about this message (depending on the *MDREP* options)

When the queue manager or a message channel agent generates a report message, it sets the *MDMID* field in the way specified by the *MDREP* field of the original message, either RONMI or ROPMI. Applications that generate report messages should also do this.

For the MQGET call, *MDMID* is one of the five fields that can be used to select a particular message to be retrieved from the queue. Normally the MQGET call returns the next message on the queue, but if a particular message is required, this can be obtained by specifying one or more of the five selection criteria, in any combination; these fields are:

- *MDMID*
- *MDCID*
- *MDGID*
- *MDSEQ*
- *MDOFF*

The application sets one or more of these field to the values required, and then sets the corresponding MO* match options in the *GMMO* field in MQGMO to indicate that those fields should be used as selection criteria. Only messages that have the specified values in those fields are candidates

⁴ An *MDMID* generated by the queue manager consists of a 4-byte product identifier (AMQ- or CSQ- in either ASCII or EBCDIC, where - represents a single blank character), followed by a product-specific implementation of a unique string. In IBM MQ this contains the first 12 characters of the queue manager name, and a value derived from the system clock. All queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, to ensure that message identifiers are unique. The ability to generate a unique string also depends upon the system clock not being changed backward. To eliminate the possibility of a message identifier generated by the queue manager duplicating one generated by the application, the application should avoid generating identifiers with initial characters in the range A through I in ASCII or EBCDIC (X'41' through X'49' and X'C1' through X'C9'). However, the application is not prevented from generating identifiers with initial characters in these ranges.

for retrieval. The default for the *GMMO* field (if not altered by the application) is to match both the message identifier and the correlation identifier.

Normally, the message returned is the *first* message on the queue that satisfies the selection criteria. But if *GMBRWN* is specified, the message returned is the *next* message that satisfies the selection criteria; the scan for this message starts with the message *following* the current cursor position.

Note: The queue is scanned sequentially for a message that satisfies the selection criteria, so retrieval times will be slower than if no selection criteria are specified, especially if many messages have to be scanned before a suitable one is found.

See [Table 1](#) for more information about how selection criteria are used in various situations.

Specifying *MINONE* as the message identifier has the same effect as *not* specifying *MOMSGI*, that is, *any* message identifier will match.

This field is ignored if the *GMMUC* option is specified in the *GMO* parameter on the *MQGET* call.

On return from an *MQGET* call, the *MDMID* field is set to the message identifier of the message returned (if any).

The following special value may be used:

MINONE

No message identifier is specified.

The value is binary zero for the length of the field.

This is an input/output field for the *MQGET*, *MQPUT*, and *MQPUT1* calls. The length of this field is given by *LN MID*. The initial value of this field is *MINONE*.

MDMT (10-digit signed integer)

Message type.

This indicates the type of the message. Message types are grouped as follows:

MTSFST

Lowest value for system-defined message types.

MTSLST

Highest value for system-defined message types.

The following values are currently defined within the system range:

MTDGRM

Message not requiring a reply.

The message is one that does not require a reply.

MTRQST

Message requiring a reply.

The message is one that requires a reply.

The name of the queue to which the reply should be sent must be specified in the *MDRQ* field. The *MDREP* field indicates how the *MDMID* and *MDCID* of the reply are to be set.

MTRPLY

Reply to an earlier request message.

The message is the reply to an earlier request message (*MTRQST*). The message should be sent to the queue indicated by the *MDRQ* field of the request message. The *MDREP* field of the request should be used to control how the *MDMID* and *MDCID* of the reply are set.

Note: The queue manager does not enforce the request-reply relationship; this is an application responsibility.

MTRPRT

Report message.

The message is reporting on some expected or unexpected occurrence, usually related to some other message (for example, a request message was received which contained data that was not valid). The message should be sent to the queue indicated by the *MDRQ* field of the message descriptor of the original message. The *MDFB* field should be set to indicate the nature of the report. The *MDREP* field of the original message can be used to control how the *MDMID* and *MDCID* of the report message should be set.

Report messages generated by the queue manager or message channel agent are always sent to the *MDRQ* queue, with the *MDFB* and *MDCID* fields set as described previously.

Other values within the system range may be defined in future versions of the MQI, and are accepted by the MQPUT and MQPUT1 calls without error.

Application-defined values can also be used. They must be within the following range:

MTAFST

Lowest value for application-defined message types.

MTALST

Highest value for application-defined message types.

For the MQPUT and MQPUT1 calls, the *MDMT* value must be within either the system-defined range or the application-defined range; if it is not, the call fails with reason code RC2029.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is MTDGRM.

MDOFF (10-digit signed integer)

Offset of data in physical message from start of logical message.

This is the offset in bytes of the data in the physical message from the start of the logical message of which the data forms part. This data is called a *segment*. The offset is in the range 0 through 999 999 999. A physical message which is not a segment of a logical message has an offset of zero.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, PMLOGO is specified.
- On the MQGET call, MOOFFS is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application does not comply with these conditions, or the call is MQPUT1, the application must ensure that *MDOFF* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 1. On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

For a report message reporting on a segment of a logical message, the *MDOLN* field (provided it is not OLUNDF) is used to update the offset in the segment information retained by the queue manager.

On input to the MQGET call, the queue manager uses the value detailed in Table 1. On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is zero. This field is ignored if *MDVER* is less than MDVER2.

MDOLN (10-digit signed integer)

Length of original message.

This field is of relevance only for report messages that are segments. It specifies the length of the message segment to which the report message relates; it does not specify the length of the logical message of which the segment forms part, nor the length of the data in the report message.

Note: When generating a report message for a message that is a segment, the queue manager and message channel agent copy into the MQMD for the report message the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL*, fields from the original message. As a result, the report message is also a segment.

Applications that generate report messages are recommended to do the same, and to ensure that the *MDOLN* field is set correctly.

The following special value is defined:

OLUNDF

Original length of message not defined.

MDOLN is an input field on the MQPUT and MQPUT1 calls, but the value provided by the application is accepted only in particular circumstances:

- If the message being put is a segment and is also a report message, the queue manager accepts the value specified. The value must be:
 - Greater than zero if the segment is not the last segment
 - Not less than zero if the segment is the last segment
 - Not less than the length of data present in the message

If these conditions are not satisfied, the call fails with reason code RC2252.

- If the message being put is a segment but not a report message, the queue manager ignores the field and uses the length of the application message data instead.
- In all other cases, the queue manager ignores the field and uses the value OLUNDF instead.

This is an output field on the MQGET call.

The initial value of this field is OLUNDF. This field is ignored if *MDVER* is less than MDVER2.

MDPAN (28-byte character string)

Name of application that put the message.

This is part of the **origin context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

The format of the *MDPAN* depends on the value of *MDPAT*.

When this field is set by the queue manager (that is, for all options except PMSETA), it is set to value which is determined by the environment:

- On z/OS, the queue manager uses:
 - For z/OS batch, the 8-character job name from the JES JOB card
 - For TSO, the 7-character TSO user identifier
 - For CICS, the 8-character applid, followed by the 4-character tranid
 - For IMS, the 8-character IMS system identifier, followed by the 8-character PSB name
 - For XCF, the 8-character XCF group name, followed by the 16-character XCF member name
 - For a message generated by a queue manager, the first 28 characters of the queue manager name
 - For distributed queuing without CICS, the 8-character jobname of the channel initiator followed by the 8-character name of the module putting to the dead-letter queue followed by an 8-character task identifier.
 - For MQSeries Java language bindings processing with IBM MQ for z/OS the 8-character jobname of the address space created for the UNIX System Services environment. Typically, this will be a TSO user identifier with a single numeric character appended.

The name or names are each padded to the right with blanks, as is any space in the remainder of the field. Where there is more than one name, there is no separator between them.

- On PC DOS, and Windows systems, the queue manager uses:
 - For a CICS application, the CICS transaction name
 - For a non-CICS application, the rightmost 28 characters of the fully-qualified name of the executable

- On IBM i, the queue manager uses the fully-qualified job name.
- On HP Integrity NonStop Server, the queue manager uses: the rightmost 28 characters of the fully-qualified name of the executable, if this is available to the queue manager, and blanks otherwise
- On UNIX systems, the queue manager uses:
 - For a CICS application, the CICS transaction name
 - For a non-CICS application, the rightmost 14 characters of the fully-qualified name of the executable if this is available to the queue manager, and blanks otherwise (for example, on AIX)
- On VSE/ESA, the queue manager uses the 8-character applid, followed by the 4-character tranid.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If PMSETA is not specified, this field is ignored on input and is an output-only field.

This is an output field for the MQGET call. The length of this field is given by LNPAN. The initial value of this field is 28 blank characters.

MDPAT (10-digit signed integer)

Type of application that put the message.

This is part of the **origin context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

MDPAT may have one of the following standard types. User-defined types can also be used but should be restricted to values in the range ATUFST through ATULST.

ATAIX

AIX application (same value as ATUNIX).

ATBRKR

Broker.

ATCICS

CICS transaction.

ATCICB

CICS bridge.

ATVSE

CICS/VSE transaction.

ATDOS

IBM MQ MQI client application on PC DOS.

ATDQM

Distributed queue manager agent.

ATGUAR

Tandem Guardian application (same value as ATNSK).

ATIMS

IMS application.

ATIMSB

IMS bridge.

ATJAVA

Java.

ATMVS

MVS or TSO application (same value as ATZOS).

ATNOTE

Lotus Notes Agent application.

ATNSK

Tandem NonStop Kernel application.

AT390

OS/390 application (same value as ATZOS).

AT400

IBM i application.

ATQM

Queue manager.

ATUNIX

UNIX application.

ATVOS

Stratus VOS application.

ATWIN

16-bit Windows application.

ATWINT

32-bit Windows application.

ATXCF

XCF.

ATZOS

z/OS application.

ATDEF

Default application type.

This is the default application type for the platform on which the application is running.

Note: The value of this constant is environment-specific.

ATUNK

Unknown application type.

This value can be used to indicate that the application type is unknown, even though other context information is present.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

The following special value can also occur:

ATNCON

No context information present in message.

This value is set by the queue manager when a message is put with no context (that is, the PMNOC context option is specified).

When a message is retrieved, *MDPAT* can be tested for this value to decide whether the message has context (it is recommended that *MDPAT* is never set to ATNCON, by an application using PMSETA, if any of the other context fields are nonblank).

ATSIB

Indicates a message originated in another IBM MQ messaging product and arrived via the SIB (Service Integration Bus) bridge.

When the queue manager generates this information as a result of an application put, the field is set to a value that is determined by the environment. Note that on IBM i, it is set to AT400; the queue manager never uses ATCICS on IBM i.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. If PMSETA is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDPAT* that was transmitted with the message if it was put to a queue. This will be the value of *MDPAT* that is kept with

the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDPAT* when the message is sent as a publication to subscribers since they provide a value to override *MDPAT* in all publications sent to them. If the message has no context, the field is set to ATNCON.

This is an output field for the MQGET call. The initial value of this field is ATNCON.

MDPD (8-byte character string)

Date when message was put.

This is part of the **origin context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

The format used for the date when this field is generated by the queue manager is:

- YYYYMMDD

where the characters represent:

YYYY

year (four numeric digits)

MM

month of year (01 through 12)

DD

day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the *MDPD* and *MDPT* fields, subject to the system clock being set accurately to GMT.

If the message was put as part of a unit of work, the date is that when the message was put, and not the date when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. The contents of the field are not checked by the queue manager, except that any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If PMSETA is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDPD* that was transmitted with the message if it was put to a queue. This will be the value of *MDPD* that is kept with the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDPD* when the message is sent as a publication to subscribers since they provide a value to override *MDPD* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by LNPDAT. The initial value of this field is 8 blank characters.

MDPER (10-digit signed integer)

Message persistence.

This indicates whether the message survives system failures and restarts of the queue manager. For the MQPUT and MQPUT1 calls, the value must be one of the following:

PEPER

Message is persistent.

This means that the message survives system failures and restarts of the queue manager. Once the message has been put, and the putter's unit of work committed (if the message is put as part of a unit of work), the message is preserved on auxiliary storage. It remains there until the message is removed from the queue, and the getter's unit of work committed (if the message is retrieved as part of a unit of work).

When a persistent message is sent to a remote queue, a store-and-forward mechanism is used to hold the message at each queue manager along the route to the destination, until the message is known to have arrived at the next queue manager.

Persistent messages cannot be placed on:

- Temporary dynamic queues
- Shared queues where the coupling facility structure level is less than three, or the coupling facility structure is not recoverable.

Persistent messages can be placed on permanent dynamic queues, predefined queues, and shared queues where the coupling facility structure level is 3, and the coupling facility is recoverable.

PENPER

Message is not persistent.

This means that the message does not normally survive system failures or restarts of the queue manager. This applies even if an intact copy of the message is found on auxiliary storage during restart of the queue manager.

In the special case of shared queues, nonpersistent messages *do* survive restarts of queue managers in the queue-sharing group, but do not survive failures of the coupling facility used to store messages on the shared queues.

PEQDEF

Message has default persistence.

- If the queue is a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed. Usually, all of the instances of a cluster queue have the same value for the *DefPersistence* attribute, although this is not mandated.

The value of *DefPersistence* is copied into the *MDPER* field when the message is placed on the destination queue. If *DefPersistence* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *local* queue manager, even if the destination queue manager is remote.

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPersistence* is copied into the *MDPER* field when the message is put. If *DefPersistence* is changed subsequently, messages that have already been put are not affected.

Both persistent and nonpersistent messages can exist on the same queue.

When replying to a message, applications should normally use for the reply message the persistence of the request message.

For an MQGET call, the value returned is either PEPER or PENPER.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is PEQDEF.

MDPRI (10-digit signed integer)

Message priority.

For the MQPUT and MQPUT1 calls, the value must be greater than or equal to zero; zero is the lowest priority. The following special value can also be used:

PRQDEF

Default priority for queue.

- If the queue is a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed. Usually, all of the instances of a cluster queue have the same value for the *DefPriority* attribute, although this is not mandated.

The value of *DefPriority* is copied into the *MDPRI* field when the message is placed on the destination queue. If *DefPriority* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *local* queue manager, even if the destination queue manager is remote.

If there is more than one definition in the queue-name resolution path, the default priority is taken from the value of this attribute in the *first* definition in the path. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPriority* is copied into the *MDPRI* field when the message is put. If *DefPriority* is changed subsequently, messages that have already been put are not affected.

The value returned by the MQGET call is always greater than or equal to zero; the value PRQDEF is never returned.

If a message is put with a priority greater than the maximum supported by the local queue manager (this maximum is given by the *MaxPriority* queue manager attribute), the message is accepted by the queue manager, but placed on the queue at the queue manager's maximum priority; the MQPUT or MQPUT1 call completes with CCWARN and reason code RC2049. However, the *MDPRI* field retains the value specified by the application which put the message.

When replying to a message, applications should normally use for the reply message the priority of the request message. In other situations, specifying PRQDEF allows priority tuning to be carried out without changing the application.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is PRQDEF.

MDPT (8-byte character string)

Time when message was put.

This is part of the **origin context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

The format used for the time when this field is generated by the queue manager is:

- HHMMSSSTH

where the characters represent (in order):

HH

hours (00 through 23)

MM

minutes (00 through 59)

SS

seconds (00 through 59; see [note](#))

T

tenths of a second (0 through 9)

H

hundredths of a second (0 through 9)

Note: If the system clock is synchronized to a very accurate time standard, it is possible on rare occasions for 60 or 61 to be returned for the seconds in *MDPT*. This happens when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the *MDPD* and *MDPT* fields, subject to the system clock being set accurately to GMT.

If the message was put as part of a unit of work, the time is that when the message was put, and not the time when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. The contents of the field are not checked by the queue manager, except that any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If PMSETA is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDPT* that was transmitted with the message if it was put to a queue. This will be the value of *MDPT* that is kept with the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDPT* when the message is sent as a publication to subscribers since they provide a value to override *MDPT* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by LNPTIM. The initial value of this field is 8 blank characters.

MDREP (10-digit signed integer)

Options for report messages.

A report message is a message about another message, used to inform an application about expected or unexpected events that relate to the original message. The *MDREP* field enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also (for both reports and replies) how the message and correlation identifiers in the report or reply message are to be set. Any or all (or none) of the following types of report message can be requested:

- Exception
- Expiration
- Confirm on arrival (COA)
- Confirm on delivery (COD)
- Positive action notification (PAN)
- Negative action notification (NAN)

If more than one type of report message is required, or other report options are needed, the values can be added together (do not add the same constant more than once).

The application that receives the report message can determine the reason the report was generated by examining the *MDFB* field in the MQMD; see the *MDFB* field for more details.

The use of report options when putting a message to a topic can cause zero, one or many report messages to be generated and sent to the application. This is because the publication message may be sent to zero, one or many subscribing applications.

Exception options: You can specify one of the following options to request an exception report message.

ROACTIVITY

Activity reports required

This report option enables an activity report to be generated, whenever a message with this report option set is processed by supporting applications.

Messages with this report option set must be accepted by any queue manager, even if they do not 'understand' the option. This allows the report option to be set on any user message, even if they are processed by previous queue managers. To achieve this, the report option is placed in the ROAUM subfield.

If a process (either a queue manager or a user process) performs an Activity on a message with ROACT set, it can choose to generate and put an activity report.

The activity report option allows the route of any message to be traced throughout a queue manager network. The report option can be specified on any current user message and instantly they can begin to calculate the route of the message through the network. If the application generating the message cannot switch on activity reports, it can be turned on by using an API crossing exit supplied by queue manager administrators.

Several conditions are applicable to activity reports:

1. The route will be less detailed if there are fewer queue managers in the network which are able to generate activity reports.
2. The activity reports may not be easily 'orderable' in order to determine the route taken.
3. The activity reports may not be able to find a route to their requested destination.

ROEXC

Exception reports required.

This type of report can be generated by a message channel agent when a message is sent to another queue manager and the message cannot be delivered to the specified destination queue. For example, the destination queue or an intermediate transmission queue might be full, or the message might be too big for the queue.

Generation of the exception report message depends on the persistence of the original message, and the speed of the message channel (normal or fast) through which the original message travels:

- For all persistent messages, and for nonpersistent messages traveling through normal message channels, the exception report is generated *only* if the action specified by the sending application for the error condition can be completed successfully. The sending application can specify one of the following actions to control the disposition of the original message when the error condition arises:

- RODLQ (this causes the original message to be placed on the dead-letter queue).
- RODISC (this causes the original message to be discarded).

If the action specified by the sending application cannot be completed successfully, the original message is left on the transmission queue, and no exception report message is generated.

- For nonpersistent messages traveling through fast message channels, the original message is removed from the transmission queue and the exception report generated *even if* the specified action for the error condition cannot be completed successfully. For example, if RODLQ is specified, but the original message cannot be placed on the dead-letter queue because (say) that queue is full, the exception report message is generated and the original message discarded.

See [Message persistence](#) for more information about normal and fast message channels.

An exception report is not generated if the application that put the original message can be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call.

Applications can also send exception reports, to indicate that a message that it has received cannot be processed (for example, because it is a debit transaction that would cause the account to exceed its credit limit).

Message data from the original message is not included with the report message.

Do not specify more than one of ROEXC, ROEXCD, and ROEXCF.

ROEXCD

Exception reports with data required.

This is the same as ROEXC, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of ROEXC, ROEXCD, and ROEXCF.

ROEXCF

Exception reports with full data required.

This is the same as ROEXC, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of ROEXC, ROEXCD, and ROEXCF.

Expiration options: You can specify one of the following options to request an expiration report message.

ROEXP

Expiration reports required.

This type of report is generated by the queue manager if the message is discarded before delivery to an application because its expiry time has passed (see the *MDEXP* field). If this option is not set, no report message is generated if a message is discarded for this reason (even if one of the ROEXC* options is specified).

Message data from the original message is not included with the report message.

Do not specify more than one of ROEXP, ROEXPD, and ROEXPF.

ROEXPD

Expiration reports with data required.

This is the same as ROEXP, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of ROEXP, ROEXPD, and ROEXPF.

ROEXPF

Expiration reports with full data required.

This is the same as ROEXP, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of ROEXP, ROEXPD, and ROEXPF.

Confirm-on-arrival options: You can specify one of the following options to request a confirm-on-arrival report message.

ROCOA

Confirm-on-arrival reports required.

This type of report is generated by the queue manager that owns the destination queue, when the message is placed on the destination queue. Message data from the original message is not included with the report message.

If the message is put as part of a unit of work, and the destination queue is a local queue, the COA report message generated by the queue manager becomes available for retrieval only if and when the unit of work is committed.

A COA report is not generated if the *MDFMT* field in the message descriptor is FMXQH or FMDLH. This prevents a COA report being generated if the message is put on a transmission queue, or is undeliverable and put on a dead-letter queue.

Do not specify more than one of ROCOA, ROCOAD, and ROCOAF.

ROCOAD

Confirm-on-arrival reports with data required.

This is the same as ROCOA, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

Do not specify more than one of ROCOA, ROCOAD, and ROCOAF.

ROCOAF

Confirm-on-arrival reports with full data required.

This is the same as ROCOA, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of ROCOA, ROCOAD, and ROCOAF.

Discard and expiry options: You can specify the following option to set the expiry time and discard flag for report messages.

ROPDAE

Set report message expiry time and discard flag.

This option ensures that report messages and reply messages inherit the expiry time and discard flag (whether to discard or not), from their original messages. With this option set, report and reply messages:

1. Inherit the RODISC flag (if it was set).
2. Inherit the remaining expiry time of the message, if the message is not an expiry report. If the message is an expiry report, the expiry time is set to 60 seconds.

With this option set, the following applies:

Note:

1. Report and reply messages are generated with a discard flag and an expiry value, and cannot remain within the system.
2. Trace route messages are prevented from reaching destination queues on non-trace route enabled queue managers.
3. Queues are prevented from being filled with reports that cannot be delivered, if communications links are broken.
4. Command server responses inherit the remaining expiry of the request.

Confirm-on-delivery options: You can specify one of the following options to request a confirm-on-delivery report message.

ROCOD

Confirm-on-delivery reports required.

This type of report is generated by the queue manager when an application retrieves the message from the destination queue in a way that causes the message to be deleted from the queue. Message data from the original message is not included with the report message.

If the message is retrieved as part of a unit of work, the report message is generated within the same unit of work, so that the report is not available until the unit of work is committed. If the unit of work is backed out, the report is not sent.

A COD report is not generated if the *MDFMT* field in the message descriptor is *FMDLH*. This prevents a COD report being generated if the message is undeliverable and put on a dead-letter queue.

ROCOD is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

ROCODD

Confirm-on-delivery reports with data required.

This is the same as ROCOD, except that the first 100 bytes of the application message data from the original message are included in the report message. If the original message contains one or more MQ header structures, they are included in the report message, in addition to the 100 bytes of application data.

If GMATM is specified on the MQGET call for the original message, and the message retrieved is truncated, the amount of application message data placed in the report message is the minimum of:

- The length of the original message
- 100 bytes.

ROCODD is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

ROCODF

Confirm-on-delivery reports with full data required.

This is the same as ROCOD, except that all of the application message data from the original message is included in the report message.

ROCODF is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

Action-notification options: You can specify one or both of the following options to request that the receiving application send a positive-action or negative-action report message.

ROPAN

Positive action notification reports required.

This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has been performed successfully. The application generating the report determines whether any data is to be included with the report.

Other than conveying this request to the application retrieving the message, the queue manager takes no action based upon this option. It is the responsibility of the retrieving application to generate the report if appropriate.

RONAN

Negative action notification reports required.

This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has *not* been performed successfully. The application generating the report determines whether any data is to be included with the report. For example, it may be desirable to include some data indicating why the request could not be performed.

Other than conveying this request to the application retrieving the message, the queue manager takes no action based upon this option. It is the responsibility of the retrieving application to generate the report if appropriate.

Determination of which conditions correspond to a positive action and which correspond to a negative action is the responsibility of the application. However, it is recommended that if the request has been only partially performed, a NAN report rather than a PAN report should be generated if requested. It is also recommended that every possible condition should correspond to either a positive action, or a negative action, but not both.

Message-identifier options: You can specify one of the following options to control how the *MDMID* of the report message (or of the reply message) is to be set.

RONMI

New message identifier.

This is the default action, and indicates that if a report or reply is generated as a result of this message, a new *MDMID* is to be generated for the report or reply message.

ROPMI

Pass message identifier.

If a report or reply is generated as a result of this message, the *MDMID* of this message is to be copied to the *MDMID* of the report or reply message.

The *MsgId* of a publication message will be different for each subscriber that receives a copy of the publication and therefore the *MsgId* copied into the report or reply message will be different for each one.

If this option is not specified, RONMI is assumed.

Correlation-identifier options: You can specify one of the following options to control how the *MDCID* of the report message (or of the reply message) is to be set.

ROCMTC

Copy message identifier to correlation identifier.

This is the default action, and indicates that if a report or reply is generated as a result of this message, the *MDMID* of this message is to be copied to the *MDCID* of the report or reply message.

The *MsgId* of a publication message will be different for each subscriber that receives a copy of the publication and therefore the *MsgId* copied into the *CorrelId* of the report or reply message will be different for each one.

ROPICI

Pass correlation identifier.

If a report or reply is generated as a result of this message, the *MDCID* of this message is to be copied to the *MDCID* of the report or reply message.

The *MDCID* of a publication message will be specific to a subscriber unless it uses the *SOSCID* option and sets the *SCDIC* field in the MQSD to CINONE. Therefore it is possible that the *MDCID* copied into the *MDCID* of the report or reply message will be different for each one.

If this option is not specified, ROCMTC is assumed.

Servers replying to requests or generating report messages are recommended to check whether the ROPMI or ROPICI options were set in the original message. If they were, the servers should take the action described for those options. If neither is set, the servers should take the corresponding default action.

: You can specify one of the following options to control the disposition of the original message when it cannot be delivered to the destination queue. These options apply only to those situations that would result in an exception report message being generated if one had been requested by the sending application. The application can set the disposition options independently of requesting exception reports.

RODLQ

Place message on dead-letter queue.

This is the default action, and indicates that the message should be placed on the dead-letter queue, if the message cannot be delivered to the destination queue. This happens in the following situations:

- When the application that put the original message cannot be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call. An exception report message is generated, if one was requested by the sender.
- When the application that put the original message was putting to a topic

An exception report message will be generated, if one was requested by the sender.

RODISC

Discard message.

This indicates that the message should be discarded if it cannot be delivered to the destination queue. This happens in the following situations:

- When the application that put the original message cannot be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call. An exception report message is generated, if one was requested by the sender.
- When the application that put the original message was putting to a topic

An exception report message will be generated, if one was requested by the sender.

If it is desired to return the original message to the sender, without the original message being placed on the dead-letter queue, the sender should specify RODISC with ROEXCF.

Default option: You can specify the following if no report options are required:

RONONE

No reports required.

This value can be used to indicate that no other options have been specified. RONONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

General information:

1. All report types required must be specifically requested by the application sending the original message. For example, if a COA report is requested but an exception report is not, a COA report is generated when the message is placed on the destination queue, but no exception report is generated if the destination queue is full when the message arrives there. If no *MDREP* options are set, no report messages are generated by the queue manager or message channel agent (MCA).

Some report options can be specified even though the local queue manager does not recognize them; this is useful when the option is to be processed by the *destination* queue manager. See [“Report options and message flags” on page 1388](#) for more details.

If a report message is requested, the name of the queue to which the report should be sent must be specified in the *MDRQ* field. When a report message is received, the nature of the report can be determined by examining the *MDFB* field in the message descriptor.

2. If the queue manager or MCA that generates a report message is unable to put the report message on the reply queue (for example, because the reply queue or transmission queue is full), the report message is placed instead on the dead-letter queue. If that *also* fails, or there is no dead-letter queue, the action taken depends on the type of the report message:
 - If the report message is an exception report, the message which caused the exception report to be generated is left on its transmission queue; this ensures that the message is not lost.
 - For all other report types, the report message is discarded and processing continues normally. This is done because either the original message has already been delivered safely (for COA or COD report messages), or is no longer of any interest (for an expiration report message).

Once a report message has been placed successfully on a queue (either the destination queue or an intermediate transmission queue), the message is no longer subject to special processing; it is treated just like any other message.

3. When the report is generated, the *MDRQ* queue is opened and the report message put using the authority of the *MDUID* in the MQMD of the message causing the report, except in the following cases:
 - Exception reports generated by a receiving MCA are put with whatever authority the MCA used when it tried to put the message causing the report. The *CDPA* channel attribute determines the user identifier used.
 - COA reports generated by the queue manager are put with whatever authority was used when the message causing the report was put on the queue manager generating the report. For example, if the message was put by a receiving MCA using the MCA's user identifier, the queue manager puts the COA report using the MCA's user identifier.

Applications generating reports should normally use the same authority as they would have used to generate a reply; this should normally be the authority of the user identifier in the original message.

If the report has to travel to a remote destination, senders and receivers can decide whether to accept it, in the same way as they do for other messages.

4. If a report message with data is requested:
 - The report message is always generated with the amount of data requested by the sender of the original message. If the report message is too big for the reply queue, the processing described previously occurs; the report message is never truncated in order to fit on the reply queue.
 - If the *MDFMT* of the original message is *FMXQH*, the data included in the report does not include the *MQXQH*. The report data starts with the first byte of the data beyond the *MQXQH* in the original message. This occurs whether the queue is a transmission queue.
5. If a COA, COD, or expiration report message is received at the reply queue, it is guaranteed that the original message arrived, was delivered, or expired, as appropriate. However, if one or more of these report messages is requested and is *not* received, the reverse cannot be assumed, since one of the following may have occurred:
 - a. The report message is held up because a link is down.
 - b. The report message is held up because a blocking condition exists at an intermediate transmission queue or at the reply queue (for example, the queue is full or inhibited for puts).
 - c. The report message is on a dead-letter queue.
 - d. When the queue manager was attempting to generate the report message, it was unable to put it on the appropriate queue, and was also unable to put it on the dead-letter queue, so the report message could not be generated.
 - e. A failure of the queue manager occurred between the action being reported (arrival, delivery or expiry), and generation of the corresponding report message. (This does not happen for COD report messages if the application retrieves the original message within a unit of work, as the COD report message is generated within the same unit of work.)

Exception report messages may be held up in the same way for reasons 1, 2, and 3 previously. However, when an MCA is unable to generate an exception report message (the report message cannot be put either on the reply queue or the dead-letter queue), the original message remains on the transmission queue at the sender, and the channel is closed. This occurs irrespective of whether the report message was to be generated at the sending or the receiving end of the channel.

6. If the original message is temporarily blocked (resulting in an exception report message being generated and the original message being put on a dead-letter queue), but the blockage clears and an application then reads the original message from the dead-letter queue and puts it again to its destination, the following may occur:

- Even though an exception report message has been generated, the original message eventually arrives successfully at its destination.
- More than one exception report message is generated in respect of a single original message, since the original message may encounter another blockage later.

Report messages when putting to a topic:

1. Reports can be generated when putting a message to a topic. This message will be sent to all subscribers to the topic, which could be zero, one or many. This should be taken into account when choosing to use report options as many report messages could be generated as a result.
2. When putting a message to a topic, there may be many destination queues that are to be given a copy of the message. If some of these destination queues have a problem, such as queue full, then the successful completion of the MQPUT depends on the setting of NPMMSGDLV or PMSGDLV (depending on the persistence of the message). If the setting is such that message delivery to the destination queue must be successful (for example, it is a persistent message to a durable subscriber and PMSGDLV is set to ALL or ALLDUR), then success is defined as one of the following criteria being met:
 - Successful put to the subscriber queue
 - Use of RODLQ and a successful put to the Dead-letter queue if the subscriber queue cannot take the message
 - Use of RODISC if the subscriber queue cannot take the message.

Report messages for message segments:

1. Report messages can be requested for messages that have segmentation allowed (see the description of the MFSEGA flag). If the queue manager finds it necessary to segment the message, a report message can be generated for each of the segments that subsequently encounters the relevant condition. Applications should therefore be prepared to receive multiple report messages for each type of report message requested. The *MDGID* field in the report message can be used to correlate the multiple reports with the group identifier of the original message, and the *MDFB* field used to identify the type of each report message.
2. If GMLOGO is used to retrieve report messages for segments, be aware that reports of *different types* may be returned by the successive MQGET calls. For example, if both COA and COD reports are requested for a message that is segmented by the queue manager, the MQGET calls for the report messages may return the COA and COD report messages interleaved in an unpredictable fashion. This can be avoided by using the GMCMPM option (optionally with GMATM). GMCMPM causes the queue manager to reassemble report messages that have the same report type. For example, the first MQGET call might reassemble all of the COA messages relating to the original message, and the second MQGET call might reassemble all of the COD messages. Which is reassembled first depends on which type of report message happens to occur first on the queue.
3. Applications that themselves put segments can specify different report options for each segment. However, the following points should be noted:
 - If the segments are retrieved using the GMCMPM option, only the report options in the *first* segment are honored by the queue manager.
 - If the segments are retrieved one by one, and most of them have one of the ROCOD* options, but at least one segment does not, it will not be possible to use the GMCMPM option to retrieve the report messages with a single MQGET call, or use the GMASGA option to detect when all of the report messages have arrived.
4. In an MQ network, it is possible for the queue managers to have differing capabilities. If a report message for a segment is generated by a queue manager or MCA that does not support segmentation, the queue manager or MCA will not by default include the necessary segment information in the report message, and this may make it difficult to identify the original message that caused the report to be generated. This difficulty can be avoided by requesting data with the report message, that is, by specifying the appropriate RO*D or RO*F options. However, be aware that if RO*D is specified, *less than* 100 bytes of application message data may be returned to the

application which retrieves the report message, if the report message is generated by a queue manager or MCA that does not support segmentation.

Contents of the message descriptor for a report message: When the queue manager or message channel agent (MCA) generates a report message, it sets the fields in the message descriptor to the following values, and then puts the message in the normal way.

Field in MQMD	Value used
<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER2
<i>MDREP</i>	RONONE
<i>MDMT</i>	MTRPRT
<i>MDEXP</i>	EIULIM
<i>MDFB</i>	As appropriate for the nature of the report (FBCOA, FBCOD, FBEXP, or an RC* value)
<i>MDENC</i>	Copied from the original message descriptor
<i>MDCSI</i>	Copied from the original message descriptor
<i>MDFMT</i>	Copied from the original message descriptor
<i>MDPRI</i>	Copied from the original message descriptor
<i>MDPER</i>	Copied from the original message descriptor
<i>MDMID</i>	As specified by the report options in the original message descriptor
<i>MDCID</i>	As specified by the report options in the original message descriptor
<i>MDBOC</i>	0
<i>MDRQ</i>	Blanks
<i>MDRM</i>	Name of queue manager
<i>MDUID</i>	As set by the PMPASI option
<i>MDACC</i>	As set by the PMPASI option
<i>MDAID</i>	As set by the PMPASI option
<i>MDPAT</i>	ATQM, or as appropriate for the message channel agent
<i>MDPAN</i>	First 28 bytes of the queue manager name or message channel agent name. For report messages generated by the IMS bridge, this field contains the XCF group name and XCF member name of the IMS system to which the message relates.
<i>MDPD</i>	Date when report message is sent
<i>MDPT</i>	Time when report message is sent
<i>MDAOD</i>	Blanks
<i>MDGID</i>	Copied from the original message descriptor
<i>MDSEQ</i>	Copied from the original message descriptor
<i>MDOFF</i>	Copied from the original message descriptor
<i>MDMFL</i>	Copied from the original message descriptor
<i>MDOLN</i>	Copied from the original message descriptor if not OLUNDF, and set to the length of the original message data otherwise

An application generating a report is recommended to set similar values, except for the following:

- The *MDRM* field can be set to blanks (the queue manager will change this to the name of the local queue manager when the message is put).
- The context fields should be set using the option that would have been used for a reply, normally PMPASI.

Analyzing the report field: The *MDREP* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report should use one of the techniques described in [“Analyzing the report field”](#) on page 1389.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is RNONE.

MDRM (48-byte character string)

Name of reply queue manager.

This is the name of the queue manager to which the reply message or report message should be sent. *MDRQ* is the local name of a queue that is defined on this queue manager.

If the *MDRM* field is blank, the local queue manager looks up the *MDRQ* name in its queue definitions. If a local definition of a remote queue exists with this name, the *MDRM* value in the transmitted message is replaced by the value of the *RemoteQMGrName* attribute from the definition of the remote queue, and this value will be returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, the *MDRM* that is transmitted with the message is the name of the local queue manager.

If the name is specified, it may contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise, however, no check is made that the name satisfies the naming rules for queue managers, or that this name is known to the sending queue manager; this is also true for the name transmitted, if the *MDRM* is replaced in the transmitted message.

If a reply-to queue is not required, it is recommended (although this is not checked) that the *MDRM* field should be set to blanks; the field should not be left uninitialized.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by LNQMNM. The initial value of this field is 48 blank characters.

MDRQ (48-byte character string)

Name of reply queue.

This is the name of the message queue to which the application that issued the get request for the message should send MTRPLY and MTRPRT messages. The name is the local name of a queue that is defined on the queue manager identified by *MDRM*. This queue should not be a model queue, although the sending queue manager does not verify this when the message is put.

For the MQPUT and MQPUT1 calls, this field must not be blank if the *MDMT* field has the value MTRQST, or if any report messages are requested by the *MDREP* field. However, the value specified (or substituted) is passed on to the application that issues the get request for the message, whatever the message type.

If the *MDRM* field is blank, the local queue manager looks up the *MDRQ* name in its own queue definitions. If a local definition of a remote queue exists with this name, the *MDRQ* value in the transmitted message is replaced by the value of the *RemoteQName* attribute from the definition of the remote queue, and this value will be returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, *MDRQ* is unchanged.

If the name is specified, it may contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise, however, no check is made that the name satisfies the naming

rules for queues; this is also true for the name transmitted, if the *MDRQ* is replaced in the transmitted message. The only check made is that a name has been specified, if the circumstances require it.

If a reply-to queue is not required, it is recommended (although this is not checked) that the *MDRQ* field should be set to blanks; the field should not be left uninitialized.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

If a message that requires a report message cannot be delivered, and the report message also cannot be delivered to the queue specified, both the original message and the report message go to the dead-letter (undelivered-message) queue (see the *DeadLetterQName* attribute described in [“Attributes for the queue manager” on page 1355](#)).

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

MDSEQ (10-digit signed integer)

Sequence number of logical message within group.

Sequence numbers start at 1, and increase by 1 for each new logical message in the group, up to a maximum of 999 999 999. A physical message which is not in a group has a sequence number of 1.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, PMLOGO is specified.
- On the MQGET call, MOSEQN is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *MDSEQ* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in [Table 1](#). On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

On input to the MQGET call, the queue manager uses the value detailed in [Table 1](#). On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is one. This field is ignored if *MDVER* is less than MDVER2.

MDSID (4-byte character string)

Structure identifier.

The value must be:

MDSIDV

Identifier for message descriptor structure.

This is always an input field. The initial value of this field is MDSIDV.

MDUID (12-byte character string)

User identifier.

This is part of the **identity context** of the message. For more information about message context, see [Message context](#) and [Controlling context information](#).

MDUID specifies the user identifier of the application that originated the message. The queue manager treats this information as character data, but does not define the format of it.

After a message has been received, *MDUID* can be used in the *ODAU* field of the *OBJDSC* parameter of a subsequent MQOPEN or MQPUT1 call, so that the authorization check is performed for the *MDUID* user instead of the application performing the open.

When the queue manager generates this information for an MQPUT or MQPUT1 call, the queue manager uses a user identifier determined from the environment.

When the user identifier is determined from the environment:

- On z/OS, the queue manager uses:
 - For batch, the user identifier from the JES JOB card or started task
 - For TSO, the log on user identifier
 - For CICS, the user identifier associated with the task
 - For IMS, the user identifier depends on the type of application:
 - For:
 - Nonmessage BMP regions
 - Nonmessage IFP regions
 - Message BMP and message IFP regions that have *not* issued a successful GU callthe queue manager uses the user identifier from the region JES JOB card or the TSO user identifier. If these are blank or null, it uses the name of the program specification block (PSB).
 - For:
 - Message BMP and message IFP regions that *have* issued a successful GU call
 - MPP regionsthe queue manager uses one of:
 - The signed-on user identifier associated with the message
 - The logical terminal (LTERM) name
 - The user identifier from the region JES JOB card
 - The TSO user identifier
 - The PSB name
- On IBM i, the queue manager uses the name of the user profile associated with the application job.
- On HP Integrity NonStop Server, the queue manager uses the MQSeries principal that is defined for the Tandem user identifier in the MQSeries principal database.
- On UNIX systems, the queue manager uses:
 - The application's logon name
 - The effective user identifier of the process if no logon is available
 - The user identifier associated with the transaction, if the application is a CICS transaction
- On VSE/ESA, this is a reserved field.
- On Windows, the queue manager uses the first 12 characters of the logged-on user name.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETI or PMSETA is specified in the *PMO* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If PMSETI or PMSETA is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDUID* that was transmitted with the message if it was put to a queue. This will be the value of *MDUID* that is kept with the message if it is retained (see description of PMRET for more details about retained publications) but is not used as the *MDUID* when the message is sent as a publication to subscribers since they provide a value to override *MDUID* in all publications sent to them. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by LNUID. The initial value of this field is 12 blank characters.

MDVER (10-digit signed integer)

Structure version number.

The value must be one of the following:

MDVER1

Version-1 message descriptor structure.

MDVER2

Version-2 message descriptor structure.

Note: When a version-2 MQMD is used, the queue manager performs additional checks on any MQ header structures that may be present at the beginning of the application message data; for further details see the usage notes for the MQPUT call.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MDVERC

Current version of message descriptor structure.

This is always an input field. The initial value of this field is MDVER1.

Initial values

Table 598. Initial values of fields in MQMD		
Field name	Name of constant	Value of constant
MDSID	MDSIDV	'MD--'
MDVER	MDVER1	1
MDREP	RONONE	0
MDMT	MTDGRM	8
MDEXP	EIULIM	-1
MDFB	FBNONE	0
MDENC	ENNAT	Depends on environment
MDCSI	CSQM	0
MDFMT	FMNONE	Blanks
MDPRI	PRQDEF	-1
MDPER	PEQDEF	2
MDMID	MINONE	Nulls
MDCID	CINONE	Nulls
MDBOC	None	0
MDRQ	None	Blanks
MDRM	None	Blanks
MDUID	None	Blanks
MDACC	ACNONE	Nulls
MDAID	None	Blanks
MDPAT	ATNCON	0
MDPAN	None	Blanks
MDPD	None	Blanks

Table 598. Initial values of fields in MQMD (continued)

Field name	Name of constant	Value of constant
MDPT	None	Blanks
MDAOD	None	Blanks
MDGID	GINONE	Nulls
MDSEQ	None	1
MDOFF	None	0
MDMFL	MFNONE	0
MDOLN	OLUNDF	-1

Notes:

1. The symbol ~ represents a single blank character.

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D*
D* MQMD Structure
D*
D* Structure identifier
D  MDSID          1      4      INZ('MD ')
D* Structure version number
D  MDVER          5      8I 0 INZ(1)
D* Options for report messages
D  MDREP          9     12I 0 INZ(0)
D* Message type
D  MDMT          13     16I 0 INZ(8)
D* Message lifetime
D  MDEXP         17     20I 0 INZ(-1)
D* Feedback or reason code
D  MDFB          21     24I 0 INZ(0)
D* Numeric encoding of message data
D  MDENC         25     28I 0 INZ(273)
D* Character set identifier of messagedata
D  MDCSI         29     32I 0 INZ(0)
D* Format name of message data
D  MDFMT         33     40      INZ(' ')
D* Message priority
D  MDPRI         41     44I 0 INZ(-1)
D* Message persistence
D  MDPER         45     48I 0 INZ(2)
D* Message identifier
D  MDMID         49     72      INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D* Correlation identifier
D  MDCID         73     96      INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D* Backout counter
D  MDBOC         97    100I 0 INZ(0)
D* Name of reply queue
D  MDRQ         101    148      INZ
D* Name of reply queue manager
D  MDRM         149    196      INZ
D* User identifier
D  MDUID        197    208      INZ
D* Accounting token
D  MDACC        209    240      INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     000000')
D* Application data relating to identity
D  MDAID        241    272      INZ
D* Type of application that put the message
D  MDPAT        273    276I 0 INZ(0)
D* Name of application that put the message

```

D	MDPAN	277	304	INZ
D*	Date when message was put			
D	MDPD	305	312	INZ
D*	Time when message was put			
D	MDPT	313	320	INZ
D*	Application data relating to origin			
D	MDAOD	321	324	INZ
D*	Group identifier			
D	MDGID	325	348	INZ(X'0000000000000000-00000000000000000000-000000000000')
D				
D*	Sequence number of logical message within group			
D	MDSEQ	349	352I 0	INZ(1)
D*	Offset of data in physical message from start of logical message			
D	MDOFF	353	356I 0	INZ(0)
D*	Message flags			
D	MDMFL	357	360I 0	INZ(0)
D*	Length of original message			
D	MDOLN	361	364I 0	INZ(-1)

MQMDE - Message descriptor extension

Overview

Purpose: The MQMDE structure describes the data that sometimes occurs preceding the application message data. The structure contains those MQMD fields that exist in the version-2 MQMD, but not in the version-1 MQMD.

Format name: FMMDE.

Character set and encoding: Data in MQMDE must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT for the C programming language.

The character set and encoding of the MQMDE must be set into the *MDCSI* and *MDENC* fields in:

- The MQMD (if the MQMDE structure is at the start of the message data), or
- The header structure that precedes the MQMDE structure (all other cases).

If the MQMDE is not in the queue manager's character set and encoding, the MQMDE is accepted but not honored, that is, the MQMDE is treated as message data.

Usage: Normal applications should use a version-2 MQMD, in which case they will not encounter an MQMDE structure. However, specialized applications, and applications that continue to use a version-1 MQMD, may encounter an MQMDE in some situations. The MQMDE structure can occur in the following circumstances:

- Specified on the MQPUT and MQPUT1 calls
- Returned by the MQGET call
- In messages on transmission queues
- [“MQMDE specified on MQPUT and MQPUT1 calls” on page 1111](#)
- [“MQMDE returned by MQGET call” on page 1112](#)
- [“MQMDE in messages on transmission queues” on page 1112](#)
- [“Fields” on page 1113](#)
- [“Initial values” on page 1115](#)
- [“RPG declaration” on page 1115](#)

MQMDE specified on MQPUT and MQPUT1 calls

On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *MDFMT* field in MQMD to FMMDE to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager

assumes default values for the fields in the MQMDE. The default values that the queue manager uses are the same as the initial values for the structure - see [Table 600 on page 1115](#).

If the application provides a version-2 MQMD *and* prefixes the application message data with an MQMDE, the structures are processed as shown in [Table 599 on page 1112](#).

<i>Table 599. Queue-manager action when MQMDE specified on MQPUT or MQPUT1</i>			
MQMD version	Values of version-2 fields	Values of corresponding fields in MQMDE	Action taken by queue manager
1	-	Valid	MQMDE is honored
2	Default	Valid	MQMDE is honored
2	Not default	Valid	MQMDE is treated as message data
1 or 2	Any	Not valid	Call fails with an appropriate reason code
1 or 2	Any	MQMDE is in the wrong character set or encoding, or is an unsupported version	MQMDE is treated as message data

There is one special case. If the application uses a version-2 MQMD to put a message that is a segment (that is, the MFSEG or MFLSEG flag is set), and the format name in the MQMD is FMDLH, the queue manager generates an MQMDE structure and inserts it *between* the MQDLH structure and the data that follows it. In the MQMD that the queue manager retains with the message, the version-2 fields are set to their default values.

Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on MQPUT and MQPUT1. However, the queue manager does *not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

MQMDE returned by MQGET call

On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a nondefault value. The queue manager sets the *MDFMT* field in MQMD to the value FMMDE to indicate that an MQMDE is present.

If the application provides an MQMDE at the start of the *BUFFER* parameter, the MQMDE is ignored. On return from the MQGET call, it is replaced by the MQMDE for the message (if one is needed), or overwritten by the application message data (if the MQMDE is not needed).

If an MQMDE is returned by the MQGET call, the data in the MQMDE is typically in the queue manager's character set and encoding. However the MQMDE may be in some other character set and encoding if:

- The MQMDE was treated as data on the MQPUT or MQPUT1 call (see [Table 599 on page 1112](#) for the circumstances that can cause this).
- The message was received from a remote queue manager connected by a TCP connection, and the receiving message channel agent (MCA) was not set up correctly (see [Security of IBM MQ for IBM i objects](#) for further information).

MQMDE in messages on transmission queues

Messages on transmission queues are prefixed with the MQXQH structure, which contains within it a version-1 MQMD. An MQMDE may also be present, positioned between the MQXQH structure and

application message data, but it will typically be present only if one or more of the fields in the MQMDE has a nondefault value.

Other IBM MQ header structures can also occur between the MQXQH structure and the application message data. For example, when the dead-letter header MQDLH is present, and the message is not a segment, the order is:

- MQXQH (containing a version-1 MQMD)
- MQMDE
- MQDLH
- Application message data

Fields

The MQMDE structure contains the following fields; the fields are described in **alphabetical order**:

MECSI (10-digit signed integer)

Character-set identifier of data that follows MQMDE.

This specifies the character set identifier of the data that follows the MQMDE structure; it does not apply to character data in the MQMDE structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that this field is valid. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

MEENC (10-digit signed integer)

MEENC (10-digit signed integer)

This specifies the numeric encoding of the data that follows the MQMDE structure; it does not apply to numeric data in the MQMDE structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that the field is valid. See the *MDENC* field described in [“MQMD - Message descriptor” on page 1068](#) for more information about data encodings.

The initial value of this field is ENNAT.

MEFLG (10-digit signed integer)

General flags.

The following flag can be specified:

MEFNON

No flags.

The initial value of this field is MEFNON.

MEFMT (8-byte character string)

Format name of data that follows MQMDE.

This specifies the format name of the data that follows the MQMDE structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The queue manager does not check that this field is valid. See the *MDFMT* field described in [“MQMD - Message descriptor”](#) on page 1068 for more information about format names.

The initial value of this field is FMNONE.

MEGID (24-byte bit string)

Group identifier.

See the *MDGID* field described in [“MQMD - Message descriptor”](#) on page 1068. The initial value of this field is GINONE.

MELEN (10-digit signed integer)

Length of MQMDE structure.

The following value is defined:

MELEN2

Length of version-2 message descriptor extension structure.

The initial value of this field is MELEN2.

MEMFL (10-digit signed integer)

Message flags.

See the *MDMFL* field described in [“MQMD - Message descriptor”](#) on page 1068. The initial value of this field is MFNONE.

MEOFF (10-digit signed integer)

Offset of data in physical message from start of logical message.

See the *MDOFF* field described in [“MQMD - Message descriptor”](#) on page 1068. The initial value of this field is 0.

MEOLN (10-digit signed integer)

Length of original message.

See the *MDOLN* field described in [“MQMD - Message descriptor”](#) on page 1068. The initial value of this field is OLUNDF.

MESEQ (10-digit signed integer)

Sequence number of logical message within group.

See the *MDSEQ* field described in [“MQMD - Message descriptor”](#) on page 1068. The initial value of this field is 1.

MESID (4-byte character string)

Structure identifier.

The value must be:

MESIDV

Identifier for message descriptor extension structure.

The initial value of this field is MESIDV.

MEVER (10-digit signed integer)

Structure version number.

The value must be:

MEVER2

Version-2 message descriptor extension structure.

The following constant specifies the version number of the current version:

MEVERC

Current version of message descriptor extension structure.

The initial value of this field is MEVER2.

Initial values

Table 600. Initial values of fields in MQMDE		
Field name	Name of constant	Value of constant
MESID	MESIDV	'MDE¬'
MEVER	MEVER2	2
MELEN	MELEN2	72
MEENC	ENNAT	Depends on environment
MECSI	CSUNDF	0
MEFMT	FMNONE	Blanks
MEFLG	MEFNON	0
MEGID	GINONE	Nulls
MESEQ	None	1
MEOFF	None	0
MEMFL	MFNONE	0
MEOLN	OLUNDF	-1
Notes:		
1. The symbol ¬ represents a single blank character.		

RPG declaration

```
D*.1....:....2....:....3....:....4....:....5....:....6....:....7..
D*
D* MQMDE Structure
D*
D* Structure identifier
D MESID          1      4    INZ('MDE ')
D* Structure version number
D MEVER          5      8I 0 INZ(2)
D* Length of MQMDE structure
D MELEN          9     12I 0 INZ(72)
D* Numeric encoding of data that followsMQMDE
D MEENC         13     16I 0 INZ(273)
D* Character-set identifier of data thatfollows MQMDE
D MECSI         17     20I 0 INZ(0)
D* Format name of data that followsMQMDE
D MEFMT         21     28    INZ('      ')
D* General flags
D MEFLG         29     32I 0 INZ(0)
D* Group identifier
D MEGID         33     56    INZ(X'0000000000000000-
D                      000000000000000000000000-
D                      000000000000')
D* Sequence number of logical messagewithin group
D MESEQ         57     60I 0 INZ(1)
D* Offset of data in physical messagefrom start of logical message
D MEOFF         61     64I 0 INZ(0)
D* Message flags
D MEMFL         65     68I 0 INZ(0)
```

```
D* Length of original message
D MEOLN          69      72I 0 INZ(-1)
```

MQMHBO - Message handle to buffer options

Structure defining the message handle to buffer options

Overview

Purpose: The MQMHBO structure allows applications to specify options that control how buffers are produced from message handles. The structure is an input parameter on the MQMHBUF call.

Character set and encoding: Data in MQMHBO must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1116](#)
- [“Initial values” on page 1117](#)
- [“RPG declaration” on page 1117](#)

Fields

The MQMHBO structure contains the following fields; the fields are described in **alphabetical order**:

MBOPT (10-digit signed integer)

Message handle to buffer options structure - MBOPT field.

These options control the action of MQMHBUF.

You must specify the following option:

MBPRRF

When converting properties from a message handle into a buffer, convert them into the MQRFH2 format.

Optionally, you can also specify the following option. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

MBDLPR

Properties that are added to the buffer are deleted from the message handle. If the call fails no properties are deleted.

This is always an input field. The initial value of this field is MBPRRF.

MBSID (10-digit signed integer)

Message handle to buffer options structure - MBSID field.

This is the structure identifier. The value must be:

MBSIDV

Identifier for message handle to buffer options structure.

This is always an input field. The initial value of this field is MBSIDV.

MBVER (10-digit signed integer)

This is the structure version number. The value must be:

MBVER1

Version number for message handle to buffer options structure.

The following constant specifies the version number of the current version:

MBVERC

Current version of message handle to buffer options structure.

This is always an input field. The initial value of this field is MBVER1.

Initial values

Table 601. Initial values of fields in MQMHBO		
Field name	Name of constant	Value of constant
MVSID	MBSIDV	'MHBO'
MBVER	MBVER1	1
MBOPT	MBPRRF	

Notes:

1. The value Null string or blanks denotes a blank character.

RPG declaration

```
D* MQMHBO Structure
D*
D*
D* Structure identifier
D  MBSID              1      4    INZ('MHBO')
D*
D* Structure version number
D  MBVER              5      8I 0  INZ(1)
D*
D* Options that control the action of MQMHBUF
D  MBOPT              9     12I 0  INZ(1)
```

MQOD - Object descriptor

The MQOD structure is used to specify an object by name.

Overview

Purpose: The following types of object are valid:

- Queue or distribution list
- Namelist
- Process definition
- Queue manager
- Topic

The structure is an input/output parameter on the MQOPEN and MQPUT1 calls.

Version: The current version of MQOD is ODVER4. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQOD that is supported by the environment, but with the initial value of the *ODVER* field set to ODVER1. To use fields that are not present in the version-1 structure, the application must set the *ODVER* field to the version number of the version required.

To open a distribution list, *ODVER* must be ODVER2 or greater.

Character set and encoding: Data in MQOD must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 1118](#)

- [“Initial values” on page 1125](#)
- [“RPG declaration” on page 1125](#)

Fields

The MQOD structure contains the following fields; the fields are described in **alphabetical order**:

ODASI (40-byte bit string)

Alternate security identifier.

This is a security identifier that is passed with the *ODAU* to the authorization service to allow appropriate authorization checks to be performed. *ODASI* is used only if:

- OOALTU is specified on the MQOPEN call, or
- PMALTU is specified on the MQPUT1 call,

and the *ODAU* field is not entirely blank up to the first null character or the end of the field.

The *ODASI* field has the following structure:

- The first byte is a binary integer containing the length of the significant data that follows; the value excludes the length byte itself. If no security identifier is present, the length is zero.
- The second byte indicates the type of security identifier that is present; the following values are possible:

SITWNT

Windows security identifier.

SITNON

No security identifier.

- The third and subsequent bytes up to the length defined by the first byte contain the security identifier itself.
- Remaining bytes in the field are set to binary zero.

The following special value may be used:

SINONE

No security identifier specified.

The value is binary zero for the length of the field.

This is an input field. The length of this field is given by LNSCID. The initial value of this field is SINONE. This field is ignored if *ODVER* is less than ODVER3.

ODAU (12-byte character string)

Alternate user identifier.

If OOALTU is specified for the MQOPEN call, or PMALTU for the MQPUT1 call, this field contains an alternate user identifier that is to be used to check the authorization for the open, in place of the user identifier that the application is currently running under. Some checks, however, are still carried out with the current user identifier (for example, context checks).

If OOALTU and PMALTU are not specified and this field is entirely blank up to the first null character or the end of the field, the open can succeed only if no user authorization is needed to open this object with the options specified.

If neither OOALTU nor PMALTU is specified, this field is ignored.

This is an input field. The length of this field is given by LNUID. The initial value of this field is 12 blank characters.

ODDN (48-byte character string)

Dynamic queue name.

This is the name of a dynamic queue that is to be created by the MQOPEN call. This is of relevance only when *ODON* specifies the name of a model queue; in all other cases *ODDN* is ignored.

The characters that are valid in the name are the same as those for *ODON*, except that an asterisk is also valid. A name that is blank (or one in which only blanks are shown before the first null character) is not valid if *ODON* is the name of a model queue.

If the last nonblank character in the name is an asterisk (*), the queue manager replaces the asterisk with a string of characters that guarantees that the name generated for the queue is unique at the local queue manager. To allow a sufficient number of characters for this, the asterisk is valid only in positions 1 through 33. There must be no characters other than blanks or a null character following the asterisk.

It is valid for the asterisk to occur in the first character position, in which case the name consists solely of the characters generated by the queue manager.

This is an input field. The length of this field is given by *LNQN*. The initial value of this field is 'AMQ.*', padded with blanks.

ODIDC (10-digit signed integer)

Number of queues that failed to open.

This is the number of queues in the distribution list that failed to open successfully. If present, this field is also set when opening a single queue which is not in a distribution list.

Note: If present, this field is set *only* if the *CMPCOD* parameter on the MQOPEN or MQPUT1 call is CCOK or CCWARN; it is *not* set if the *CMPCOD* parameter is CCFAIL.

This is an output field. The initial value of this field is 0. This field is ignored if *ODVER* is less than ODVER2.

ODKDC (10-digit signed integer)

Number of local queues opened successfully.

This is the number of queues in the distribution list that resolve to local queues and that were opened successfully. The count does not include queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). If present, this field is also set when opening a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is ignored if *ODVER* is less than ODVER2.

ODMN (48-byte character string)

Object queue manager name.

This is the name of the queue manager on which the *ODON* object is defined. The characters that are valid in the name are the same as those for *ODON* (see previously). A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected (the local queue manager).

The following points apply to the types of object indicated:

- If *ODOT* is OTTOP, OTNLST, OTPRO, or OTQM, *ODMN* must be blank or the name of the local queue manager.
- If *ODON* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ODMN* field the name of the queue manager on which the queue is created; this is the name of the local queue manager. A model queue can be specified only on the MQOPEN call; a model queue is not valid on the MQPUT1 call.
- If *ODON* is the name of a cluster queue, and *ODMN* is blank, the actual destination of messages sent using the queue handle returned by the MQOPEN call is chosen by the queue manager (or cluster workload exit, if one is installed) as follows:

- If OOBND0 is specified, the queue manager selects an instance of the cluster queue during the processing of the MQOPEN call, and all messages put using this queue handle are sent to that instance.
- If OOBNDN is specified, the queue manager may choose a different instance of the destination queue (residing on a different queue manager in the cluster) for each successive MQPUT call that uses this queue handle.

If the application needs to send a message to a *specific* instance of a cluster queue (that is, a queue instance that resides on a particular queue manager in the cluster), the application should specify the name of that queue manager in the *ODMN* field. This forces the local queue manager to send the message to the specified destination queue manager.

- If the object being opened is a distribution list (that is, *ODREC* is greater than zero), *ODMN* must be blank or the null string. If this condition is not satisfied, the call fails with reason code RC2153.

This is an input/output field for the MQOPEN call when *ODON* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by *LNQMN*. The initial value of this field is 48 blank characters.

ODON (48-byte character string)

Object name.

This is the local name of the object as defined on the queue manager identified by *ODMN*. The name can contain the following characters:

- Uppercase alphabetic characters (A - Z)
- Lowercase alphabetic characters (a - z)
- Numeric digits (0 - 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but may contain trailing blanks. A null character can be used to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply in the environments indicated:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- On IBM i, names containing lowercase characters, forward slash, or percent, must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified for names that occur as fields in structures or as parameters on calls.

The following points apply to the types of object indicated:

- If *ODON* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ODON* field the name of the queue created. A model queue can be specified only on the MQOPEN call; a model queue is not valid on the MQPUT1 call.
- If the object being opened is a distribution list (that is, *ODREC* is present and greater than zero), *ODON* must be blank or the null string. If this condition is not satisfied, the call fails with reason code RC2152.
- If *ODOT* is OTQM, special rules apply; in this case the name must be entirely blank up to the first null character or the end of the field.
- If *ODON* is the name of an alias queue with *TARGETYPE(TOPIC)*, a security check is first made on the named alias queue, as is normal for the use of alias queues. If this security check is successful, this MQOPEN call will continue and behaves like an MQOPEN of an OTTOP, including making a security check against the administrative topic object.

This is an input/output field for the MQOPEN call when *ODON* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by *LNQN*. The initial value of this field is 48 blank characters.

The full topic name can be built from two different fields: *ODON* and *ODOS*. For details of how these two fields are used, see [“Using topic strings” on page 554](#).

ODORO (10-digit signed integer)

Offset of first object record from start of MQOD.

This is the offset in bytes of the first MQOR object record from the start of the MQOD structure. The offset can be positive or negative. *ODORO* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

When a distribution list is being opened, an array of one or more MQOR object records must be provided in order to specify the names of the destination queues in the distribution list. This can be done in one of two ways:

- By using the offset field *ODORO*

In this case, the application should declare its own structure containing an MQOD followed by the array of MQOR records (with as many array elements as are needed), and set *ODORO* to the offset of the first element in the array from the start of the MQOD. Care must be taken to ensure that this offset is correct.

- By using the pointer field *ODORP*

In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ODORP* to the address of the array.

Whichever technique is chosen, one of *ODORO* and *ODORP* must be used; the call fails with reason code RC2155 if both are zero, or both are nonzero.

This is an input field. The initial value of this field is 0. This field is ignored if *ODVER* is less than *ODVER2*.

ODORP (pointer)

Address of first object record.

This is the address of the first MQOR object record. *ODORP* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

This is an input field. The initial value of this field is the null pointer. Either *ODORP* or *ODORO* can be used to specify the object records, but not both; see the description of the *ODORO* field previously for details. If *ODORP* is not used, it must be set to the null pointer or null bytes. This field is ignored if *ODVER* is less than *ODVER2*.

ODOS (MQCHARV)

ODOS specifies the long object name to be used.

This field is referenced only for certain values of *ODOT*. See the description of *ODOT* for details of which values indicate that this field is used.

If *ODOS* is specified incorrectly, according to the description of how to use the *MQCHARV* structure, or if it exceeds the maximum length, the call fails with reason code RC2441.

This is an input field. The initial values of the fields in this structure are the same as those in the *MQCHARV* structure.

The full topic name can be built from two different fields: *ODON* and *ODOS*. For details of how these two fields are used, see [“Using topic strings” on page 554](#). This field is ignored if *ODVER* is less than *ODVER4*.

ODOT (10-digit signed integer)

Object type.

Type of object being named in *ODON*. Possible values are:

OTQ

Queue. The name of the object is found in *ODON*.

OTNLST

Namelist. The name of the object is found in *ODON*.

OTPRO

Process definition. The name of the object is found in *ODON*.

OTQM

Queue manager. The name of the object is found in *ODON*.

OTTOP

Topic. The full topic name can be built from two different fields: *ODON* and *ODOS*.

For details of how those two fields are used, see [“Using topic strings” on page 554](#).

If the object identified by the *ODON* field cannot be found, the call will fail with reason code RC2425 even if there is a string specified in *ODOS*.

This is always an input field. The initial value of this field is OTQ.

ODREC (10-digit signed integer)

Number of object records present.

This is the number of MQOR object records that have been provided by the application. If this number is greater than zero, it indicates that a distribution list is being opened, with *ODREC* being the number of destination queues in the list. It is valid for a distribution list to contain only one destination.

The value of *ODREC* must not be less than zero, and if it is greater than zero *ODOT* must be OTQ; the call fails with reason code RC2154 if these conditions are not satisfied.

This is an input field. The initial value of this field is 0. This field is ignored if *ODVER* is less than ODVER2.

ODRMN (48-byte character string)

Resolved queue manager name.

This is the name of the destination queue manager after name resolution has been performed by the local queue manager. The name returned is the name of the queue manager that owns the queue identified by *ODRQN*. *ODRMN* can be the name of the local queue manager.

If *ODRQN* is a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *ODRMN* is the name of the queue-sharing group. If the queue is owned by some other queue-sharing group, *ODRQN* can be the name of the queue-sharing group or the name of a queue manager that is a member of the queue-sharing group (the nature of the value returned is determined by the queue definitions that exist at the local queue manager).

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *ODRMN* is set to blanks:

- Not a queue
- A queue, but not opened for browse, input, or output
- A cluster queue with OOBNDN specified (or with OOBNDQ in effect when the *DefBind* queue attribute has the value BNDNOT)
- A distribution list

This is an output field. The length of this field is given by LNQN. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is ignored if *ODVER* is less than ODVER3.

ODRO (MQCHARV)

ODRO is the long object name after the queue manager resolves the name provided in *ODON*.

This field is returned only for certain types of objects, topics, and queue aliases which reference a topic object.

If the long object name is provided in *ODOS* and nothing is provided in *ODON*, the value returned in this field is the same as provided in *ODOS*.

If this field is omitted (that is *ODRO.VSBufSize* is zero), the *ODRO* is not returned, but the length is returned in *ODRO.VSLength*. If the length is shorter than the full *ODRO* then it is truncated and returns as many of the rightmost characters as can fit in the provided length.

If *ODRO* is specified incorrectly, according to the description of how to use the *MQCHARV* structure, or if it exceeds the maximum length, the call fails with reason code RC2520. This field is ignored if *ODVER* is less than *ODVER4*.

ODRQN (48-byte character string)

Resolved queue name.

This is the name of the destination queue after name resolution has been performed by the local queue manager. The name returned is the name of a queue that exists on the queue manager identified by *ODRMN*.

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *ODRQN* is set to blanks:

- Not a queue
- A queue, but not opened for browse, input, or output
- A distribution list
- An alias queue that references a topic object (refer to “*ODRO (MQCHARV)*” on page 1122 instead)

This is an output field. The length of this field is given by *LNQN*. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is ignored if *ODVER* is less than *ODVER3*.

ODRRO (10-digit signed integer)

Offset of first response record from start of *MQOD*.

This is the offset in bytes of the first *MQRR* response record from the start of the *MQOD* structure. The offset can be positive or negative. *ODRRO* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

When a distribution list is being opened, an array of one or more *MQRR* response records can be provided in order to identify the queues that failed to open (*RRCC* field in *MQRR*), and the reason for each failure (*RRREA* field in *MQRR*). The data is returned in the array of response records in the same order as the queue names occur in the array of object records. The queue manager sets the response records only when the outcome of the call is mixed (that is, some queues were opened successfully while others failed, or all failed but for differing reasons); reason code RC2136 from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *REASON* parameter of the *MQOPEN* or *MQPUT1* call, and the response records are not set. Response records are optional, but if they are supplied there must be *ODREC* of them.

The response records can be provided in the same way as the object records, either by specifying an offset in *ODRRO*, or by specifying an address in *ODRRP*; see the description of *ODORO* previously for details of how to do this. However, no more than one of *ODRRO* and *ODRRP* can be used; the call fails with reason code RC2156 if both are nonzero.

For the *MQPUT1* call, these response records are used to return information about errors that occur when the message is sent to the queues in the distribution list, as well as errors that occur when the queues are opened. The completion code and reason code from the put operation for a queue replace those from the open operation for that queue only if the completion code from the latter was CCOK or CCWARN.

This is an input field. The initial value of this field is 0. This field is ignored if *ODVER* is less than *ODVER2*.

ODRRP (pointer)

Address of first response record.

This is the address of the first MQRR response record. *ODRRP* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

Either *ODRRP* or *ODRR0* can be used to specify the response records, but not both; see the previous description of the *ODRR0* field for details. If *ODRRP* is not used, it must be set to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer. This field is ignored if *ODVER* is less than *ODVER2*.

ODSID (4-byte character string)

Structure identifier.

The value must be:

ODSIDV

Identifier for object descriptor structure.

This is always an input field. The initial value of this field is *ODSIDV*.

ODSS (MQCHARV)

ODSS contains the string used to provide the selection criteria used when retrieving messages off a queue.

ODSS must not be provided in the following cases:

- If *ODOT* is not OTQ
- If the queue being opened is not being opened using one of the input options, *OOINP**

If *ODSS* is provided in these cases, the call fails with reason code RC2516.

If *ODSS* is specified incorrectly, according to the description of how to use the *MQCHARV* structure, or if it exceeds the maximum length, the call fails with reason code RC2519. This field is ignored if *ODVER* is less than *ODVER4*.

ODUDC (10-digit signed integer)

Number of remote queues opened successfully

This is the number of queues in the distribution list that resolve to remote queues and that were opened successfully. If present, this field is also set when opening a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is ignored if *ODVER* is less than *ODVER2*.

ODVER (10-digit signed integer)

Structure version number.

The value must be one of the following:

ODVER1

Version-1 object descriptor structure.

ODVER2

Version-2 object descriptor structure.

ODVER3

Version-3 object descriptor structure.

ODVER4

Version-4 object descriptor structure.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

ODVERC

Current version of object descriptor structure.

This is always an input field. The initial value of this field is ODVER1.

Initial values

Table 602. Initial values of fields in MQOD		
Field name	Name of constant	Value of constant
ODSID	ODSIDV	'OD¬¬'
ODVER	ODVER1	1
ODOT	OTQ	1
ODON	None	Blanks
ODMN	None	Blanks
ODDN	None	'AMQ.*'
ODAU	None	Blanks
ODREC	None	0
ODKDC	None	0
ODUDC	None	0
ODIDC	None	0
ODORO	None	0
ODRRO	None	0
ODORP	None	Null pointer or null bytes
ODRRP	None	Null pointer or null bytes
ODASI	SINONE	Nulls
ODRQN	None	Blanks
ODRMN	None	Blanks
ODOS	As defined for MQCHARV	As defined for MQCHARV
ODRO	As provided in ODOS	As provided in ODOS
ODSS	None	Blanks
Notes: 1. The symbol ¬ represents a single blank character.		

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D*
D* MQOD Structure
D*
D*
D* Structure identifier
D  ODSID          1      4    INZ('OD ')
D*
D* Structure version number
D  ODVER          5      8I 0 INZ(1)
D*
D* Object type
D  ODOT          9     12I 0 INZ(1)
D*
D* Object name

```

```

D ODON 13 60 INZ
D*
D* Object queue manager name
D ODMN 61 108 INZ
D*
D* Dynamic queue name
D ODDN 109 156 INZ('AMQ.*')
D*
D* Alternate user identifier
D ODAU 157 168 INZ
D*
** Number of object records
D* present
D ODREC 169 172I 0 INZ(0)
D*
** Number of local queues opened
D* successfully
D ODKDC 173 176I 0 INZ(0)
D*
** Number of remote queues opened
D* successfully
D ODUDC 177 180I 0 INZ(0)
D*
** Number of queues that failed to
D* open
D ODIDC 181 184I 0 INZ(0)
D*
** Offset of first object record
D* from start of MQOD
D ODORO 185 188I 0 INZ(0)
D*
** Offset of first response record
D* from start of MQOD
D ODRRO 189 192I 0 INZ(0)
D*
D* Address of first object record
D ODORP 193 208* INZ(*NULL)
D*
** Address of first response
D* record
D ODRRP 209 224* INZ(*NULL)
D*
D* Alternate security identifier
D ODASI 225 264 INZ(X'0000000000000000-
D 000000000000000000000000-
D 000000000000000000000000-
D 000000000000')
D*
D* Resolved queue name
D ODRQN 265 312 INZ
D*
D* Resolved queue manager name
D ODRMN 313 360 INZ
D*
D* reserved field
D ODRE1 361 364I 0 INZ(0)
D*
D* reserved field
D ODRS2 365 368I 0 INZ(0)
D*
D* Object long name
D* Address of variable length string
D ODOSCHRP 369 384* INZ(*NULL)
D* Offset of variable length string
D ODOSCHRO 385 388I 0 INZ(0)
D* Size of buffer
D ODOSVSBS 389 392I 0 INZ(-1)
D* Length of variable length string
D ODOSCHRL 393 396I 0 INZ(0)
D* CCSID of variable length string
D ODOSCHRC 397 400I 0 INZ(-3)
D*
D* Message Selector
D* Address of variable length string
D ODSSCHRP 401 416* INZ(*NULL)
D* Offset of variable length string
D ODSSCHRO 417 420I 0 INZ(0)
D* Size of buffer
D ODSSVSBS 421 424I 0 INZ(-1)
D* Length of variable length string
D ODSSCHRL 425 428I 0 INZ(0)
D* CCSID of variable length string

```

```

D  ODSSCHRC          429      432I 0 INZ(-3)
D*
D* Resolved long object name
D* Address of variable length string
D  ODRSOCHRP          433      448*   INZ(*NULL)
D* Offset of variable length string
D  ODRSOCHRO          449      452I 0 INZ(0)
D* Size of buffer
D  ODRSOVSBS          453      456I 0 INZ(-1)
D* Length of variable length string
D  ODRSOCHRL          457      460I 0 INZ(0)
D* CCSID of variable length string
D  ODRSOCHRC          461      464I 0 INZ(-3)
D*
D* Alias queue resolved object type
D  ODRT               465      468I 0 INZ(0)

```

MQOR - Object record

The MQOR structure is used to specify the queue name and queue manager name of a single destination queue.

Overview

Purpose: MQOR is an input structure for the MQOPEN and MQPUT1 calls.

Character set and encoding: Data in MQOR must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQOPEN call, it is possible to open a list of queues; this list is called a *distribution list*. Each message put using the queue handle returned by that MQOPEN call is placed on each of the queues in the list, if the queue was opened successfully.

- [“Fields” on page 1127](#)
- [“Initial values” on page 1128](#)
- [“RPG declaration” on page 1128](#)

Fields

The MQOR structure contains the following fields; the fields are described in **alphabetical order**:

ORMN (48-byte character string)

Object queue manager name.

This is the same as the *ODMN* field in the MQOD structure (see MQOD for details).

This is always an input field. The initial value of this field is 48 blank characters.

ORON (48-byte character string)

Object name.

This is the same as the *ODON* field in the MQOD structure (see MQOD for details), except that:

- It must be the name of a queue.
- It must not be the name of a model queue.

This is always an input field. The initial value of this field is 48 blank characters.

Initial values

Table 603. Initial values of fields in MQOR		
Field name	Name of constant	Value of constant
ORON	None	Blanks
ORMN	None	Blanks

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D*  
D* MQOR Structure  
D*  
D* Object name  
D  ORON                1      48    INZ  
D* Object queue manager name  
D  ORMN                49     96    INZ
```

MQPD - Property descriptor

The **MQPD** is used to define the attributes of a property.

Overview

Purpose: The structure is an input/output parameter on the MQSETMP call and an output parameter on the MQINQMP call.

Character set and encoding: Data in MQPD must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1128](#)
- [“Initial values” on page 1131](#)
- [“RPG declaration” on page 1131](#)

Fields

The MQPD structure contains the following fields; the fields are described in **alphabetical order**:

PDCT (10-digit signed integer)

This describes what message context the property belongs to.

When a queue manager receives a message containing an IBM MQ-defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *PDCT* field.

The following option can be specified:

PDUSC

The property is associated with the user context.

No special authorization is required to be able to set a property associated with the user context using the MQSETMP call.

On a IBM WebSphere MQ 7.0 queue manager, a property associated with the user context is saved as described for OOSAVA. An MQPUT call with PMPASA specified, causes the property to be copied from the saved context into the new message.

If the option previously described is not required, the following option can be used:

PDNOC

The property is not associated with a message context.

An unrecognized value is rejected with a *PDREA* code of RC2482.

This is an input/output field to the MQSETMP call and an output field from the MQINQMP call. The initial value of this field is PDNOC.

PDCPYOPT (10-digit signed integer)

This describes which type of messages the property should be copied into.

This is an output only field for recognized IBM MQ-defined properties; IBM MQ sets the appropriate value.

When a queue manager receives a message containing an IBM MQ-defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *CopyOptions* field.

You can specify one or more of these options. To specify more than one option, either add the values together (do not add the same constant more than once), or combine the values using the bitwise OR operation (if the programming language supports bit operations).

COPFOR

This property is copied into a message being forwarded.

COPPUB

This property is copied into the message received by a subscriber when a message is being published.

COPREP

This property is copied into a reply message.

COPRP

This property is copied into a report message.

COPALL

This property is copied into all types of subsequent messages.

COPNON

This property is not copied into a message.

Default option: The following option can be specified to supply the default set of copy options:

COPDEF

This property is copied into a message being forwarded, into a report message, or into a message received by a subscriber when a message is being published.

This is equivalent to specifying the combination of options COPFOR, plus COPRP, plus COPPUB.

If none of the options described previously are required, use the following option:

COPNON

Use this value to indicate that no other copy options have been specified; programmatically no relationship exists between this property and subsequent messages. This is always returned for message descriptor properties.

This is an input/output field to the MQSETMP call and an output field from the MQINQMP call. The initial value of this field is COPDEF.

PDOPT (10-digit signed integer)

The value must be:

PDNONE

No options specified

This is always an input field. The initial value of this field is PDNONE.

PDSID (10-digit signed integer)

This is the structure identifier; the value must be:

PSIDV

Identifier for property descriptor structure.

This is always an input field. The initial value of this field is **PSIDV**.

PDSUP (10-digit signed integer)

This field describes what level of support for the message property is required of the queue manager, in order for the message containing this property to be put to a queue. This applies only to IBM MQ-defined properties; support for all other properties is optional.

The field is automatically set to the correct value when the IBM MQ-defined property is known by the queue manager. If the property is not recognized, PDSUPO is assigned. When a queue manager receives a message containing an IBM MQ-defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the *PDSUP* field.

When setting an IBM MQ-defined property using the MQSETMP call on a message handle where the CMNOVA option was set, *PDSUP* becomes an input field. This allows an application to put an IBM MQ-defined property, with the correct value, where the property is unsupported by the connected queue manager, but where the message is intended to be processed on another queue manager.

The value PDSUPO is always assigned to properties that are not IBM MQ-defined properties.

If a IBM WebSphere MQ 7.0 queue manager, that supports message properties, receives a property that contains an unrecognized *PDSUP* value, the property is treated as if:

- PDSUPR was specified if any of the unrecognized values are contained in the PDRUM.
- PDSUPL was specified if any of the unrecognized values are contained in the PDAUXM
- PDSUPO was specified otherwise.

One of the following values is returned by the MQINQMP call, or one of the values can be specified, when using the MQSETMP call on a message handle where the CMNOVA option is set:

PDSUPO

The property is accepted by a queue manager even if it is not supported. The property can be discarded in order for the message to flow to a queue manager that does not support message properties. This value is also assigned to properties that are not IBM MQ-defined.

PDSUPR

Support for the property is required. The message is rejected by a queue manager that does not support the IBM MQ-defined property. The MQPUT or MQPUT1 call fails with completion code CCFAIL and reason code RC2490.

PDSUPL

The message is rejected by a queue manager that does not support the IBM MQ-defined property if the message is destined for a local queue. The MQPUT or MQPUT1 call fails with completion code CCFAIL and reason code RC2490.

The MQPUT or MQPUT1 call succeeds if the message is destined for a remote queue manager.

This is an output field on the MQINQMP call and an input field on the MQSETMP call if the message handle was created with the CMNOVA option set. The initial value of this field is PDSUPO.

PDVER (10-digit signed integer)

This is the structure version number; the value must be:

PDVER1

Version-1 property descriptor structure.

The following constant specifies the version number of the current version:

PDVERC

Current version of property descriptor structure.

This is always an input field. The initial value of this field is **PDVER1**.

Initial values

Table 604. Initial values of fields in MQPD		
Field name	Name of constant	Value of constant
PDSID	PDSIDV	'PD'
PDVER	PDVER1	1
PDOPT	PDNONE	0
PDSUP	PDSUPO	0
PDCT	PDNOC	0
PDCPYOPT	COPDEF	0

RPG declaration

```
D* MQDMHO Structure
D*
D*
D* Structure identifier
D  DMSID              1      4      INZ('DMHO')
D*
D* Structure version number
D  DMVER              5      8I 0 INZ(1)
D*
D* Options that control the action of MQDLTMH
D  DMOPT              9      12I 0 INZ(0)
```

MQPMO - Put-message options

The MQPMO structure allows the application to specify options that control how messages are placed on queues or published to topics.

Overview

Purpose

The structure is an input/output parameter on the MQPUT and MQPUT1 calls.

Version

The current version of MQPMO is PMVER2. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow.

The COPY file provided contains the most recent version of MQPMO that is supported by the environment, but with the initial value of the *PMVER* field set to PMVER1. To use fields that are not present in the version-1 structure, the application must set the *PMVER* field to the version number of the version required.

Character set and encoding

Data in MQPMO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 1131](#)
- [“Initial values” on page 1145](#)
- [“RPG declaration” on page 1145](#)

Fields

The MQPMO structure contains the following fields; the fields are described in alphabetical order:

PMCT (10 digit signed integer)

Object handle of input queue.

If PMPASI or PMPASA is specified, this field must contain the input queue handle from which context information to be associated with the message being put is taken.

If PMPASI and PMPASA are not specified, this field is ignored.

This is an input field. The initial value of this field is 0.

PMIDC (10 digit signed integer)

Number of messages that could not be sent.

This is the number of messages that could not be sent to queues in the distribution list. The count includes queues that failed to open, and queues that were opened successfully but for which the put operation failed. This field is also set when putting a message to a single queue which is not in a distribution list.

Note: This field is set only if the *CMPCOD* parameter on the MQPUT or MQPUT1 call is CCOK or CCWARN; it is not set if the *CMPCOD* parameter is CCFAIL.

This is an output field. The initial value of this field is 0. This field is not set if *PMVER* is less than *PMVER2*.

PMKDC (10 digit signed integer)

Number of messages sent successfully to local queues.

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that are local queues. The count does not include messages sent to queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). This field is also set when putting a message to a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *PMVER* is less than *PMVER2*.

PMOPT (10 digit signed integer)

Options that control the action of MQPUT and MQPUT1.

Any or none of the following can be specified. If more than one is required the values can be added (do not add the same constant more than once). Combinations that are not valid are noted; any other combinations are valid.

Publishing options: The following options control the way messages are published to a topic.

PMSRTO

Any information filled into the MDRQ and MDRM fields of the MQMD of this publication is not passed on to subscribers. If this option is used with a report option that requires a ReplyToQ, the call fails with RC2027 .

PMRET

The publication being sent is to be retained by the queue manager. This allows a subscriber to request a copy of this publication after the time it was published, by using the MQSUBRQ call. It also allows a publication to be sent to applications which make their subscription after the time this publication was made, unless they choose not to be sent it by using the option SONEWP. If an application is sent a publication which was retained, it is indicated by the mq.IsRetained message property of that publication.

Only one publication can be retained at each node of the topic tree. That means if there already is a retained publication for this topic, published by any other application, it is replaced with this publication. It is therefore better to avoid having more than one publisher retaining messages on the same topic.

When retained publications are requested by a subscriber, the subscription used may contain a wildcard in the topic, in which case a number of retained publications might match (at various nodes in the topic tree) and several publications may be sent to the requesting application. See the description of the “MQSUBRQ - Subscription request” on page 765 call for more details.

If this option is used and the publication cannot be retained, the message is not published and the call fails with RC2479 .

Syncpoint options: The following options relate to the participation of the MQPUT or MQPUT1 call within a unit of work:

PMSYP

Put message with syncpoint control.

The request is to operate within the normal unit-of-work protocols. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted.

If this option and PMNSYP are not specified, the put request is not within a unit of work.

PMSYP must not be specified with PMNSYP.

PMNSYP

Put message without syncpoint control.

The request is to operate outside the normal unit-of-work protocols. The message is available immediately, and it cannot be deleted by backing out a unit of work.

If this option and PMSYP are not specified, the put request is not within a unit of work.

PMNSYP must not be specified with PMSYP.

Message-identifier and correlation-identifier options: The following options request the queue manager to generate a new message identifier or correlation identifier:

PMNMID

Generate a new message identifier.

This option causes the queue manager to replace the contents of the *MDMID* field in MQMD with a new message identifier. This message identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

This option can also be specified when the message is being put to a distribution list; see the description of the *PRMID* field in the MQPMR structure for details.

Using this option relieves the application of the need to reset the *MDMID* field to MINONE before each MQPUT or MQPUT1 call.

PMNCID

Generate a new correlation identifier.

This option causes the queue manager to replace the contents of the *MDCID* field in MQMD with a new correlation identifier. This correlation identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

This option can also be specified when the message is being put to a distribution list; see the description of the *PRCID* field in the MQPMR structure for details.

PMNCID is useful in situations where the application requires a unique correlation identifier.

Group and segment options: The following option relates to the processing of messages in groups and segments of logical messages. These definitions might be of help in understanding the option:

Physical message

This is the smallest unit of information that can be placed on or removed from a queue; it often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor (MQMD). Generally, physical messages

are distinguished by differing values for the message identifier (*MDMID* field in MQMD), although this is not enforced by the queue manager.

Logical message

This is a single unit of application information. In the absence of system constraints, a logical message would be the same as a physical message. But where logical messages are large, system constraints may make it advisable or necessary to split a logical message into two or more physical messages, called *segments*.

A logical message that has been segmented consists of two or more physical messages that have the same nonnull group identifier (*MDGID* field in MQMD), and the same message sequence number (*MDSEQ* field in MQMD). The segments are distinguished by differing values for the segment offset (*MDOFF* field in MQMD), which gives the offset of the data in the physical message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message typically have differing message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a nonnull group identifier, although in this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages for which segmentation has been inhibited by the sending application have a null group identifier (GINONE), unless the logical message belongs to a message group.

Message group

This is a set of one or more logical messages that have the same nonnull group identifier. The logical messages in the group are distinguished by differing values for the message sequence number, which is an integer in the range 1 through n, where n is the number of logical messages in the group. If one or more of the logical messages is segmented, there are more than n physical messages in the group.

PMLOGO

Messages in groups and segments of logical messages are put in logical order.

This option tells the queue manager how the application puts messages in groups and segments of logical messages. It can be specified only on the MQPUT call; it is not valid on the MQPUT1 call.

If PMLOGO is specified, it indicates that the application uses successive MQPUT calls to:

- Put the segments in each logical message in the order of increasing segment offset, starting from 0, with no gaps.
- Put all of the segments in one logical message before putting the segments in the next logical message.
- Put the logical messages in each message group in the order of increasing message sequence number, starting from 1, with no gaps.
- Put all of the logical messages in one message group before putting logical messages in the next message group.

This order is called "logical order".

Because the application has told the queue manager how it puts messages in groups and segments of logical messages, the application does not have to maintain and update the group and segment information about each MQPUT call, as the queue manager does this. Specifically, it means that the application does not need to set the *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD, as the queue manager sets these to the appropriate values. The application need set only the *MDMFL* field in MQMD, to indicate when messages belong to groups or are segments of logical messages, and to indicate the last message in a group or last segment of a logical message.

Once a message group or logical message has been started, subsequent MQPUT calls must specify the appropriate MF* flags in *MDMFL* in MQMD. If the application tries to put a message not in a group when there is an unterminated message group, or put a message which is not a segment when there is an unterminated logical message, the call fails with reason code RC2241 or RC2242, as appropriate. However, the queue manager retains the information about the

current message group or current logical message, and the application can terminate them by sending a message (possibly with no application message data) specifying MFLMIG or MFLSEG as appropriate, before reissuing the MQPUT call to put the message that is not in the group or not a segment.

Table 605 on page 1135 shows the combinations of options and flags that are valid, and the values of the *MDGID*, *MDSEQ*, and *MDOFF* fields that the queue manager uses in each case. Combinations of options and flags that are not shown in the table are not valid. The columns in the table have the following meanings:

LOG ORD

Indicates whether the PMLOGO option is specified on the call.

MIG

Indicates whether the MFMIG or MFLMIG option is specified on the call.

SEG

Indicates whether the MFSEG or MFLSEG option is specified on the call.

SEG OK

Indicates whether the MFSEGA option is specified on the call.

Cur grp

Indicates whether a current message group exists before the call.

Cur log msg

Indicates whether a current logical message exists before the call.

Other columns

Show the values that the queue manager uses. "Previous" denotes the value used for the field in the previous message for the queue handle.

PMRLOC

Specifies that the PMRQN in the MQPMO structure must be completed with the name of the local queue which the message actually gets put to. The ResolvedQMGrName is similarly completed with the name of the local queue manager hosting the local queue. See OORLOQ for what this means. If a user is authorized for a put to a queue then they have the required authority to specify this flag on the MQPUT call. No special authority is needed.

Table 605. MQPUT options relating to messages in groups and segments of logical messages								
Options you specify				Group and log-msg status before call		Values the queue manager uses		
LOG ORD	MIG	SEG	SEG OK	Cur grp	Cur log msg	MDGID	MDSEQ	MDOFF
Yes	No	No	No	No	No	GINONE	1	0
Yes	No	No	Yes	No	No	New group id	1	0
Yes	No	Yes	Yes or No	No	No	New group id	1	0
Yes	No	Yes	Yes or No	No	Yes	Previous group id	1	Previous offset + previous segment length
Yes	Yes	Yes or No	Yes or No	No	No	New group id	1	0
Yes	Yes	Yes or No	Yes or No	Yes	No	Previous group id	Previous sequence number + 1	0

Table 605. MQPUT options relating to messages in groups and segments of logical messages (continued)

Options you specify				Group and log-msg status before call		Values the queue manager uses		
Yes	Yes	Yes	Yes or No	Yes	Yes	Previous group id	Previous sequence number	Previous offset + previous segment length
No	No	No	No	Yes or No	Yes or No	GINONE	1	0
No	No	No	Yes	Yes or No	Yes or No	New group id if GINONE, else value in field	1	0
No	No	Yes	Yes or No	Yes or No	Yes or No	New group id if GINONE, else value in field	1	Value in field
No	Yes	No	Yes or No	Yes or No	Yes or No	New group id if GINONE, else value in field	Value in field	0
No	Yes	Yes	Yes or No	Yes or No	Yes or No	New group id if GINONE, else value in field	Value in field	Value in field

Note:

- PMLOGO is not valid on the MQPUT1 call.
- For the *MDMID* field, the queue manager generates a new message identifier if PMNMID or MINONE is specified, and uses the value in the field otherwise.
- For the *MDCID* field, the queue manager generates a new correlation identifier if PMNCID is specified, and uses the value in the field otherwise.

When PMLOGO is specified, the queue manager requires that all messages in a group and segments in a logical message be put with the same value in the *MDPER* field in MQMD, that is, all must be persistent, or all must be nonpersistent. If this condition is not satisfied, the MQPUT call fails with reason code RC2185 .

The PMLOGO option affects units of work as follows:

- If the first physical message in a group or logical message is put within a unit of work, all of the other physical messages in the group or logical message must be put within a unit of work, if the same queue handle is used. However, they need not be put within the same unit of work. This allows a message group or logical message consisting of many physical messages to be split across two or more consecutive units of work for the queue handle.
- If the first physical message in a group or logical message is not put within a unit of work, none of the other physical messages in the group or logical message can be put within a unit of work, if the same queue handle is used.

If these conditions are not satisfied, the MQPUT call fails with reason code RC2245 .

When PMLOGO is specified, the MQMD supplied on the MQPUT call must not be less than MDVER2. If this condition is not satisfied, the call fails with reason code RC2257 .

If PMLOGO is not specified, messages in groups and segments of logical messages can be put in any order, and it is not necessary to put complete message groups or complete logical messages.

It is the responsibility of the application to ensure that the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* fields have appropriate values.

This is the technique that can be used to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the *MDGID*, *MDSEQ*, *MDOFF*, *MDMFL*, and *MDPER* fields to the appropriate values, and then issue the MQPUT call with PMSYP or PMNSYP set as *necessary*, but without specifying PMLOGO. If this call is successful, the queue manager retains the group and segment information, and subsequent MQPUT calls using that queue handle can specify PMLOGO as normal.

The group and segment information that the queue manager retains for the MQPUT call is separate from the group and segment information that it retains for the MQGET call.

For any given queue handle, the application is free to mix MQPUT calls that specify PMLOGO with MQPUT calls that do not, but the following points should be noted:

- If PMLOGO is not specified, each successful MQPUT call causes the queue manager to set the group and segment information for the queue handle to the values specified by the application; this replaces the existing group and segment information retained by the queue manager for the queue handle.
- If PMLOGO is not specified, the call does not fail if there is a current message group or logical message; the call might however succeed with a CCWARN completion code. [Table 606 on page 1137](#) shows the various cases that can arise. In these cases, if the completion code is not CCOK, the reason code is one of the following (as appropriate):
 - RC2241
 - RC2242
 - RC2185
 - RC2245

Note: The queue manager does not check the group and segment information for the MQPUT1 call.

Table 606. Outcome when MQPUT or MQCLOSE call is not consistent with group and segment information

Current call is	Previous call was MQPUT with PMLOGO	Previous call was MQPUT without PMLOGO
MQPUT with PMLOGO	CCFAIL	CCFAIL
MQPUT without PMLOGO	CCWARN	CCOK
MQCLOSE with an unterminated group or logical message	CCWARN	CCOK

Applications that simply want to put messages and segments in logical order are recommended to specify PMLOGO, as this is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications may need more control than provided by the PMLOGO option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* fields in MQMD are set correctly, before each MQPUT or MQPUT1 call.

For example, an application that wants to forward physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, must not specify PMLOGO. There are two reasons for this:

- If the messages are retrieved and put in order, specifying PMLOGO causes a new group identifier to be assigned to the messages, and this might make it difficult or impossible for the originator of the messages to correlate any reply or report messages that result from the message group.

- In a complex network with multiple paths between sending and receiving queue managers, the physical messages might arrive out of order. By not specifying PMLOGO and the corresponding GMLOGO on the MQGET call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without needing to wait for the next one in logical order to arrive.

Applications that generate report messages for messages in groups or segments of logical messages must also not specify PMLOGO when putting the report message.

PMLOGO can be specified with any of the other PM* options.

Context options: The following options control the processing of message context:

PMNOC

No context is to be associated with the message.

Both identity and origin context are set to indicate no context. This means that the context fields in MQMD are set to:

- Blanks for character fields
- Nulls for byte fields
- Zeros for numeric fields

PMDEFC

Use default context.

The message is to have default context information associated with it, for both identity and origin. The queue manager sets the context fields in the message descriptor as follows:

Field in MQMD	Value used
<i>MDUID</i>	Determined from the environment if possible; set to blanks otherwise.
<i>MDACC</i>	Determined from the environment if possible; set to ACNONE otherwise.
<i>MDAID</i>	Set to blanks.
<i>MDPAT</i>	Determined from the environment.
<i>MDPAN</i>	Determined from the environment if possible; set to blanks otherwise.
<i>MDPD</i>	Set to date when message is put.
<i>MDPT</i>	Set to time when message is put.
<i>MDAOD</i>	Set to blanks.

For more information about message context, see [Message context](#) and [Controlling context information](#).

This is the default action if no context options are specified.

PMPASI

Pass identity context from an input queue handle.

The message is to have context information associated with it. Identity context is taken from the queue handle specified in the *PMCT* field. Origin context information is generated by the queue manager in the same way that it is for PMDEFC (see the previous table for values). For more information about message context, see [Message context](#) and [Controlling context information](#).

For the MQPUT call, the queue must have been opened with the OOPASI option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the OOPASI option.

PMPASA

Pass all context from an input queue handle.

The message is to have context information associated with it. Both identity and origin context are taken from the queue handle specified in the *PMCT* field. For more information about message context, see [Message context](#) and [Controlling context information](#).

For the MQPUT call, the queue must have been opened with the OOPASA option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the OOPASA option.

PMSETI

Set identity context from the application.

The message is to have context information associated with it. The application specifies the identity context in the MQMD structure. Origin context information is generated by the queue manager in the same way that it is for PMDEFC (see the previous table for values). For more information about message context, see [Message context](#) and [Controlling context information](#).

For the MQPUT call, the queue must have been opened with the OOSETI option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the OOSETI option.

PMSETA

Set all context from the application.

The message is to have context information associated with it. The application specifies the identity and origin context in the MQMD structure. For more information about message context, see [Message context](#) and [Controlling context information](#).

For the MQPUT call, the queue must have been opened with the OOSETA option. For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the OOSETA option.

Only one of the PM* context options can be specified. If none of these options is specified, PMDEFC is assumed.

Put response types. The following options control the response returned to an MQPUT or MQPUT1 call. You can only specify one of these options. If PMARES and PMSRES are not specified, PMRASQ or PMRAST is assumed.

PMARES

The PMARES option requests that an MQPUT or MQPUT1 operation is completed without the application waiting for the queue manager to complete the call. Using this option can improve messaging performance, particularly for applications using client bindings. An application can periodically check, using the MQSTAT verb, whether an error has occurred during any previous asynchronous calls.

With this option, only the following fields are guaranteed to be completed in the MQMD;

- MDAID
- MDPAT
- MDPAN
- MDAOD

Additionally, if either or both of PMNMID or PMNCID are specified as options, the MDMID and MDCID returned are also completed. (PMNMID can be implicitly specified by specifying a blank MDMID field).

Only the fields previously specified are completed. Other information that would normally be returned in the MQMD or MQPMO structure is undefined.

When requesting asynchronous put response for MQPUT or MQPUT1, a CMPCOD and REASON of CCOK and RCNONE does not necessarily mean that the message was successfully put to a queue. When developing an MQI application that uses asynchronous put response and require confirmation that messages have been put to a queue you should check both CMPCOD and REASON codes from the put operations and also use MQSTAT to query asynchronous error information.

Although the success or failure of each individual MQPUT/MQPUT1 call might not be returned immediately, the first error that occurred under an asynchronous call can be determined at a later juncture through a call to MQSTAT.

If a persistent message under syncpoint fails to be delivered using asynchronous put response, and you attempt to commit the transaction, the commit fails and the transaction is backed out with a completion code of CCFAIL and a reason of RC2003 . The application can make a call to MQSTAT to determine the cause of a previous MQPUT or MQPUT1 failure

PMSRES

Specifying this value for a put option in the MQPMO structure ensures that the MQPUT or MQPUT1 operation is always issued synchronously. If the operation is successful, all fields in the MQMD and MQPMO are completed. It is provided to ensure a synchronous response irrespective of the default put response value defined on the queue or topic object.

PMRASQ

If this value is specified for an MQPUT call, the put response type used is taken from the DEFPRESP value specified on the queue when it was opened by the application. If a client application is connected to a queue manager at a level earlier than Version 7.0, it behaves as if PMSRES was specified.

If this option is specified for an MQPUT1 call, the DEFPRESP value from the queue definition is not used. If the MQPUT1 call is using PMSYP it behaves as for PMARES, and if it is using PMNSYP it behaves as for PMSRES.

PMRAST

This is a synonym for PMRASQ for use with topic objects.

Other options: The following options control authorization checking, and what happens when the queue manager is quiescing:

PMALTU

Validate with specified user identifier.

This indicates that the *ODAU* field in the *OBJDSC* parameter of the MQPUT1 call contains a user identifier that is to be used to validate authority to put messages on the queue. The call can succeed only if this *ODAU* is authorized to open the queue with the specified options, regardless of whether the user identifier under which the application is running is authorized to do so. (This does not apply to the context options specified, however, which are always checked against the user identifier under which the application is running.)

This option is valid only with the MQPUT1 call.

PMFIQ

Fail if queue manager is quiescing.

This option forces the MQPUT or MQPUT1 call to fail if the queue manager is in the quiescing state.

The call returns completion code CCFAIL with reason code RC2161 .

Default option: If none of the options described previously are required, the following option can be used:

PMNONE

No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. PMNONE is defined to aid program documentation; it is not intended that this option is used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *PMOPT* field is PMNONE.

PMPRF (10 digit signed integer)

Flags indicating which MQPMR fields are present.

This field contains flags that must be set to indicate which MQPMR fields are present in the put message records provided by the application. *PMPRF* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero, or both *PMPRO* and *PMPRP* are zero.

For fields that are present, the queue manager uses for each destination the values from the fields in the corresponding put message record. For fields that are absent, the queue manager uses the values from the MQMD structure.

One or more of the following flags can be specified to indicate which fields are present in the put message records:

PFMID

Message-identifier field is present.

PFCID

Correlation-identifier field is present.

PFGID

Group-identifier field is present.

PFFB

Feedback field is present.

PFACC

Accounting-token field is present.

If this flag is specified, either *PMSETI* or *PMSETA* must be specified in the *PMOPT* field; if this condition is not satisfied, the call fails with reason code RC2158 .

If no MQPMR fields are present, the following can be specified:

PFNONE

No put-message record fields are present.

If this value is specified, either *PMREC* must be zero, or both *PMPRO* and *PMPRP* must be zero.

PFNONE is defined to aid program documentation. It is not intended that this constant is used with any other, but as its value is zero, such use cannot be detected.

If *PMPRF* contains flags which are not valid, or put message records are provided but *PMPRF* has the value PFNONE, the call fails with reason code RC2158 .

This is an input field. The initial value of this field is PFNONE. This field is ignored if *PMVER* is less than *PMVER2*.

MMPRO (10 digit signed integer)

Offset of first put message record from start of MQPMO.

This is the offset in bytes of the first MQPMR put message record from the start of the MQPMO structure. The offset can be positive or negative. *PMPRO* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

When the message is being put to a distribution list, an array of one or more MQPMR put message records can be provided in order to specify certain properties of the message for each destination individually; these properties are:

- message identifier
- correlation identifier
- group identifier
- feedback value
- accounting token

It is not necessary to specify all of these properties, but whatever subset is chosen, the fields must be specified in the correct order. See the description of the MQPMR structure for further details.

Usually, there should be as many put message records as there are object records specified by MQOD when the distribution list is opened; each put message record supplies the message properties for

the queue identified by the corresponding object record. Queues in the distribution list which fail to open must still have put message records allocated for them at the appropriate positions in the array, although the message properties are ignored in this case.

It is possible for the number of put message records to differ from the number of object records. If there are fewer put message records than object records, the message properties for the destinations which do not have put message records are taken from the corresponding fields in the message descriptor MQMD. If there are more put message records than object records, the excess are not used (although it must still be possible to access them). Put message records are optional, but if they are supplied there must be *PMREC* of them.

The put message records can be provided in a similar way to the object records in MQOD, either by specifying an offset in *PMPRO*, or by specifying an address in *PMPRP* ; for details of how to do this, see the *ODORO* field described in [“MQOD - Object descriptor” on page 1117](#).

No more than one of *PMPRO* and *PMPRP* can be used; the call fails with reason code RC2159 if both are nonzero.

This is an input field. The initial value of this field is 0. This field is ignored if *PMVER* is less than *PMVER2*.

PMPRP (pointer)

Address of first put message record.

This is the address of the first MQPMR put message record. *PMPRP* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

Either *PMPRP* or *PMPRO* can be used to specify the put message records, but not both; see the description of the [PMRRO](#) field for details. If *PMPRP* is not used, it must be set to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer. This field is ignored if *PMVER* is less than *PMVER2*.

PMREC (10 digit signed integer)

Number of put message records or response records present.

This is the number of MQPMR put message records or MQRR response records that have been provided by the application. This number can be greater than zero only if the message is being put to a distribution list. Put message records and response records are optional - the application need not provide any records, or it can choose to provide records of only one type. However, if the application provides records of both types, it must provide *PMREC* records of each type.

The value of *PMREC* need not be the same as the number of destinations in the distribution list. If too many records are provided, the excess are not used; if too few records are provided, default values are used for the message properties for those destinations that do not have put message records (see *PMPRO* later in this topic).

If *PMREC* is less than zero, or is greater than zero but the message is not being put to a distribution list, the call fails with reason code RC2154 .

This is an input field. The initial value of this field is 0. This field is ignored if *PMVER* is less than *PMVER2*.

PMRMN (48 byte character string)

Resolved name of destination queue manager.

This is the name of the destination queue manager after name resolution has been performed by the local queue manager. The name returned is the name of the queue manager that owns the queue identified by *PMRQN*, and can be the name of the local queue manager.

If *PMRQN* is a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *PMRMN* is the name of the queue-sharing group. If the queue is owned by some other queue-sharing group, *PMRQN* can be the name of the queue-sharing group or the name of a

queue manager that is a member of the queue-sharing group (the nature of the value returned is determined by the queue definitions that exist at the local queue manager).

A nonblank value is returned only if the object is a single queue; if the object is a distribution list or topic, the value returned is undefined.

This is an output field. The length of this field is given by LNQMNM. The initial value of this field is 48 blank characters.

PMRQN (48 byte character string)

Resolved name of destination queue.

This is the name of the destination queue after name resolution has been performed by the local queue manager. The name returned is the name of a queue that exists on the queue manager identified by *PMRMN*.

A nonblank value is returned only if the object is a single queue; if the object is a distribution list or topic, the value returned is undefined.

This is an output field. The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

PMRRO (10 digit signed integer)

Offset of first response record from start of MQPMO.

This is the offset in bytes of the first MQRR response record from the start of the MQPMO structure. The offset can be positive or negative. *PMRRO* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

When the message is being put to a distribution list, an array of one or more MQRR response records can be provided in order to identify the queues to which the message was not sent successfully (*RRCC* field in MQRR), and the reason for each failure (*RRREA* field in MQRR). The message might not have been sent either because the queue failed to open, or because the put operation failed. The queue manager sets the response records only when the outcome of the call is mixed (that is, some messages were sent successfully while others failed, or all failed but for differing reasons); reason code RC2136 from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *REASON* parameter of the MQPUT or MQPUT1 call, and the response records are not set.

Usually, there should be as many response records as there are object records specified by MQOD when the distribution list is opened; when necessary, each response record is set to the completion code and reason code for the put to the queue identified by the corresponding object record. Queues in the distribution list which fail to open must still have response records allocated for them at the appropriate positions in the array, although they are set to the completion code and reason code resulting from the open operation, rather than the put operation.

It is possible for the number of response records to differ from the number of object records. If there are fewer response records than object records, it may not be possible for the application to identify all of the destinations for which the put operation failed, or the reasons for the failures. If there are more response records than object records, the excess are not used (although it must still be possible to access them). Response records are optional, but if they are supplied there must be *PMREC* of them.

The response records can be provided in a similar way to the object records in MQOD, either by specifying an offset in *PMRRO*, or by specifying an address in *PMRRP*; for details of how to do this, see the *ODORO* field described in “MQOD - Object descriptor” on page 1117. However, no more than one of *PMRRO* and *PMRRP* can be used; the call fails with reason code RC2156 if both are nonzero.

For the MQPUT1 call, this field must be zero. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is 0. This field is ignored if *PMVER* is less than *PMVER2*.

PMRRP (pointer)

Address of first response record.

This is the address of the first MQRR response record. *PMRRP* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

Either *PMRRP* or *PMRRO* can be used to specify the response records, but not both; see the description of the *PMRRO* field for details. If *PMRRP* is not used, it must be set to the null pointer or null bytes.

For the MQPUT1 call, this field must be the null pointer or null bytes. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is the null pointer. This field is ignored if *PMVER* is less than *PMVER2*.

PMSID (4 byte character string)

Structure identifier.

The value must be:

PMSIDV

Identifier for put-message options structure.

This is always an input field. The initial value of this field is PMSIDV.

PMSL (MQLONG)

The level of subscription targeted by this publication.

Only those subscriptions with the highest *PMSL* less than or equal to this value receives this publication. This value must be in the range zero to 9; zero is the lowest level.

The initial value of this field is 9.

PMTO (10 digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is -1.

PMUDC (10 digit signed integer)

Number of messages sent successfully to remote queues.

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that resolve to remote queues. Messages that the queue manager retains temporarily in distribution-list form count as the number of individual destinations that those distribution lists contain. This field is also set when putting a message to a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *PMVER* is less than *PMVER2*.

PMVER (10 digit signed integer)

Structure version number.

The value must be one of the following:

PMVER1

Version-1 put-message options structure.

PMVER2

Version-2 put-message options structure.

Fields that exist only in the more-recent version of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

PMVERC

Current version of put-message options structure.

This is always an input field. The initial value of this field is PMVER1.

Initial values

Table 607. Initial values of fields in MQPMO		
Field name	Name of constant	Value of constant
PMSID	PMSIDV	'PMO '
PMVER	PMVER1	1
PMOPT	PMNONE	0
PMT0	None	-1
PMCT	None	0
PMKDC	None	0
PMUDC	None	0
PMIDC	None	0
PMRQN	None	Blanks
PMRMN	None	Blanks
PMREC	None	0
PMPRF	PFNONE	0
PMPRO	None	0
PMRRO	None	0
PMPRP	None	Null pointer or null bytes
PMRRP	None	Null pointer or null bytes
Note:		
1. The symbol <code> </code> represents a single blank character.		

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..
D* MQPMO Structure
D*
D* Structure identifier
D PMSID          1      4      INZ('PMO ')
D* Structure version number
D PMVER          5      8I 0 INZ(1)
D* Options that control the action of MQPUT and MQPUT1
D PMOPT          9      12I 0 INZ(0)
D* Reserved
D PMT0          13     16I 0 INZ(-1)
D* Object handle of input queue
D PMCT          17     20I 0 INZ(0)
D* Number of messages sent successfully to local queues
D PMKDC          21     24I 0 INZ(0)
D* Number of messages sent successfully to remote queues
D PMUDC          25     28I 0 INZ(0)
D* Number of messages that could not be sent
D PMIDC          29     32I 0 INZ(0)
D* Resolved name of destination queue
D PMRQN          33     80      INZ
D* Resolved name of destination queue manager
```

D	PMRMN	81	128	INZ
D*	Number of put message records or response records present			
D	PMREC	129	132I 0	INZ(0)
D*	Flags indicating which MQPMR fields are present			
D	PMPRF	133	136I 0	INZ(0)
D*	Offset of first put message record from start of MQPMO			
D	PMPRO	137	140I 0	INZ(0)
D*	Offset of first response record from start of MQPMO			
D	PMRRO	141	144I 0	INZ(0)
D*	Address of first put message record			
D	PMPRP	145	160*	INZ(*NULL)
D*	Address of first response record			
D	PMRRP	161	176*	INZ(*NULL)
D*	Original message handle			
D	PMOMH	177	184I 0	
D*	New message handle			
D	PMNMH	185	190I 0	
D*	The action being performed			
D	PMACT	191	194I 0	
D*	Reserved			
D	PMRE1	195	198I 0	

MQPMR - Put-message record

The MQPMR structure is used to specify various message properties for a single destination when a message is being put to a distribution list.

Overview

Purpose: MQPMR is an input/output structure for the MQPUT and MQPUT1 calls.

Character set and encoding: Data in MQPMR must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQPUT or MQPUT1 call, it is possible to specify different values for each destination queue in a distribution list. Some of the fields are input only, others are input/output.

Note: This structure is unusual in that it does not have a fixed layout. The fields in this structure are optional, and the presence or absence of each field is indicated by the flags in the *PMPRF* field in MQPMO. Fields that are present **must occur in the following order** :

- *PRMID*
- *PRCID*
- *PRGID*
- *PRFB*
- *PRACC*

Fields that are absent occupy no space in the record.

Because MQPMR does not have a fixed layout, no definition of it is provided in the COPY file. The application programmer should create a declaration containing the fields that are required by the application, and set the flags in *PMPRF* to indicate the fields that are present.

- [“Fields” on page 1146](#)
- [“Initial values” on page 1148](#)
- [“RPG declaration” on page 1148](#)

Fields

The MQPMR structure contains the following fields; the fields are described in **alphabetical order**:

PRACC (32-byte bit string)

Accounting token.

This is the accounting token to be used for the message sent to the queue with a name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MDACC* field in MQMD for a put to a single queue. See the description of *MDACC* in [“MQMD - Message descriptor” on page 1068](#) for information about the content of this field.

If this field is not present, the value in MQMD is used.

This is an input field.

PRCID (24-byte bit string)

Correlation identifier.

This is the correlation identifier to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MDCID* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *PRCID* field.

If PMNCID is specified, a *single* new correlation identifier is generated and used for all of the destinations in the distribution list, regardless of whether they have MQPMR records. This is different from the way that PMNMID is processed (see *PRMID* field).

This is an input/output field.

PRFB (10-digit signed integer)

Feedback or reason code.

This is the feedback code to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MDFB* field in MQMD for a put to a single queue.

If this field is not present, the value in MQMD is used.

This is an input field.

PRGID (24-byte bit string)

Group identifier.

This is the group identifier to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MDGID* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *PRGID* field. The value is processed as documented in [Table 605 on page 1135](#), but with the following differences:

- In those cases where a new group identifier would be used, the queue manager generates a different group identifier for each destination (that is, no two destinations have the same group identifier).
- In those cases where the value in the field would be used, the call fails with reason code RC2258.

This is an input/output field.

PRMID (24-byte bit string)

Message identifier.

This is the message identifier to be used for the message sent to the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MDMID* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *PRMID* field. If that value is MINONE, a new message identifier is generated for *each* of those destinations (that is, no two of those destinations have the same message identifier).

If PMNMID is specified, new message identifiers are generated for all of the destinations in the distribution list, regardless of whether they have MQPMR records. This is different from the way that PMNCID is processed (see *PRCID* field).

This is an input/output field.

Initial values

There are no initial values defined for this structure, as no structure declaration is provided. The following sample declaration shows how the structure should be declared by the application programmer if all of the fields are required.

RPG declaration

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQPMR Structure
D*
D* Message identifier
D  PRMID                1      24
D* Correlation identifier
D  PRCID                25     48
D* Group identifier
D  PRGID                49     72
D* Feedback or reason code
D  PRFB                73     76I 0
D* Accounting token
D  PRACC               77     108
```

MQRFH - Rules and formatting header

The MQRFH structure defines the layout of the rules and formatting header.

Overview

Purpose: This header can be used to send string data in the form of name/value pairs.

Format name: FMRFH.

Character set and encoding: The fields in the MQRFH structure (including *RFNVS*) are in the character set and encoding given by the *MDCSI* and *MDENC* fields in the header structure that precedes the MQRFH, or by those fields in the MQMD structure if the MQRFH is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

- [“Fields” on page 1148](#)
- [“Initial values” on page 1150](#)
- [“RPG declaration” on page 1151](#)

Fields

The MQRFH structure contains the following fields; the fields are described in **alphabetical order**:

RFCSI (10-digit signed integer)

Character set identifier of data that follows *RFNVS*.

This specifies the character set identifier of the data that follows *RFNVS* ; it does not apply to character data in the MQRFH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

Numeric encoding of data that follows *RFNVS*.

This specifies the numeric encoding of the data that follows *RFNVS* ; it does not apply to numeric data in the MQRFH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is ENNAT.

RFFLG (10-digit signed integer)

Flags.

The following can be specified:

RFNONE

No flags.

The initial value of this field is RFNONE.

RFFMT (8-byte character string)

Format name of data that follows *RFNVS*.

This specifies the format name of the data that follows *RFNVS*.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

The initial value of this field is FMNONE.

RFLEN (10-digit signed integer)

Total length of MQRFH including *RFNVS*.

This is the length in bytes of the MQRFH structure, including the *RFNVS* field at the end of the structure. The length does *not* include any user data that follows the *RFNVS* field.

To avoid problems with data conversion of the user data in some environments, consider using *RFLEN* as a multiple of four.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *RFNVS* field:

RFLENV

Length of fixed part of MQRFH structure.

The initial value of this field is RFLENV.

RFNVS (n-byte character string)

String containing name/value pairs.

This is a variable-length character string containing name/value pairs in the form:

```
name1 value1 name2 value2 name3 value3 ...
```

Each name or value must be separated from the adjacent name or value by one or more blank characters; these blanks are not significant. A name or value can contain significant blanks by prefixing and suffixing the name or value with the quotation mark character; all characters between the opening quotation mark and the matching closing quotation mark are treated as significant. In the following example, the name is FAMOUS_WORDS, and the value is Hello World:

```
FAMOUS_WORDS "Hello World"
```

A name or value can contain any characters other than the null character (which acts as a delimiter for *RFNVS*). However, to assist interoperability an application might prefer to restrict names to the following characters:

- First character: uppercase or lowercase alphabetic (A through Z, or a through z), or underscore.
- Subsequent characters: upper or lowercase alphabetic, decimal digit (0 through 9), underscore, hyphen, or dot.

If a name or value contains one or more quotation marks, the name or value must be enclosed in quotation marks, and each quotation mark within the string must be doubled:

```
Famous_Words "The program displayed ""Hello World"""
```

Names and values are case sensitive, that is, lowercase letters are not considered to be the same as uppercase letters. For example, FAMOUS_WORDS and Famous_Words are two different names.

The length in bytes of *RFNVS* is equal to *RFLEN* minus *RFLENV*. To avoid problems with data conversion of the user data in some environments, it is recommended that this length should be a multiple of four. *RFNVS* must be padded with blanks to this length, or terminated earlier by placing a null character following the last significant character in the string. The null character and the bytes following it, up to the specified length of *RFNVS*, are ignored.

Note: Because the length of this field is not fixed, the field is omitted from the declarations of the structure that are provided for the supported programming languages.

RFSID (4-byte character string)

Structure identifier.

The value must be:

RFSIDV

Identifier for rules and formatting header structure.

The initial value of this field is RFSIDV.

RFVER (10-digit signed integer)

Structure version number.

The value must be:

RFVER1

Version-1 rules and formatting header structure.

The initial value of this field is RFVER1.

Initial values

Table 608. Initial values of fields in MQRFH		
Field name	Name of constant	Value of constant
RFSID	RFSIDV	'RFH↵'

Table 608. Initial values of fields in MQRFH (continued)

Field name	Name of constant	Value of constant
RFVER	RFVER1	1
RFLEN	RFLENV	32
RFENC	ENNAT	Depends on environment
RFCSI	CSUNDF	0
RFFMT	FMNONE	Blanks
RFFLG	RFNONE	0
Notes: 1. The symbol ~ represents a single blank character.		

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D* MQRFH Structure
D*
D* Structure identifier
D  RFSID          1      4    INZ('RFH ')
D* Structure version number
D  RFVER          5      8I 0 INZ(1)
D* Total length of MQRFH includingNameValueString
D  RFLEN          9     12I 0 INZ(32)
D* Numeric encoding of data that followsNameValueString
D  RFENC         13     16I 0 INZ(273)
D* Character set identifier of data thatfollows NameValueString
D  RFCSI         17     20I 0 INZ(0)
D* Format name of data that followsNameValueString
D  RFFMT         21     28    INZ(' ')
D* Flags
D  RFFLG         29     32I 0 INZ(0)

```

MQRFH2 - Rules and formatting header 2

The MQRFH2 structure defines the format of the version-2 rules and formatting header.

Overview

Purpose: This header can be used to send data that has been encoded using an XML-like syntax. A message can contain two or more MQRFH2 structures in series, with user data optionally following the last MQRFH2 structure in the series.

Format name: FMRFH2.

Character set and encoding: Special rules apply to the character set and encoding used for the MQRFH2 structure:

- Fields other than *RF2NVD* are in the character set and encoding given by the *MDCSI* and *MDENC* fields in the header structure that precedes MQRFH2, or by those fields in the MQMD structure if the MQRFH2 is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

When GMCONV is specified on the MQGET call, the queue manager converts these fields to the requested character set and encoding.

- RF2NVD* is in the character set given by the *RF2NVC* field. Only certain Unicode character sets are valid for *RF2NVC* (see the description of *RF2NVC* for details).

Some character sets have a representation that is dependent on the encoding. If *RF2NVC* is one of these character sets, *RF2NVD* must be in the same encoding as the other fields in the *MQRFH2*.

When *GMCONV* is specified on the *MQGET* call, the queue manager converts *RF2NVD* to the requested encoding, but does not change its character set.

- [“Fields” on page 1152](#)
- [“Initial values” on page 1156](#)
- [“RPG declaration” on page 1156](#)

Fields

The *MQRFH2* structure contains the following fields; the fields are described in **alphabetical order**:

RF2CSI (10-digit signed integer)

Character set identifier of data that follows last *RF2NVD* field.

This specifies the character set identifier of the data that follows the last *RF2NVD* field; it does not apply to character data in the *MQRFH2* structure itself.

On the *MQPUT* or *MQPUT1* call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value *CSINHT* is not returned by the *MQGET* call.

CSINHT cannot be used if the value of the *MDPAT* field in *MQMD* is *ATBRKR*.

The initial value of this field is *CSINHT*.

RF2ENC (10-digit signed integer)

Numeric encoding of data that follows last *RF2NVD* field.

This specifies the numeric encoding of the data that follows the last *RF2NVD* field; it does not apply to numeric data in the *MQRFH2* structure itself.

On the *MQPUT* or *MQPUT1* call, the application must set this field to the value appropriate to the data.

The initial value of this field is *ENNAT*.

RF2FLG (10-digit signed integer)

Flags.

The following value must be specified:

RFNONE

No flags.

The initial value of this field is *RFNONE*.

RF2FMT (8-byte character string)

Format name of data that follows last *RF2NVD* field.

This specifies the format name of the data that follows the last *RF2NVD* field.

On the *MQPUT* or *MQPUT1* call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in *MQMD*.

The initial value of this field is *FMNONE*.

RF2LEN (10-digit signed integer)

Total length of MQRFH2 including all *RF2NVL* and *RF2NVD* fields.

This is the length in bytes of the MQRFH2 structure, including the *RF2NVL* and *RF2NVD* fields at the end of the structure. It is valid for there to be multiple pairs of *RF2NVL* and *RF2NVD* fields at the end of the structure, in the sequence:

```
length1, data1, length2, data2, ...
```

RF2LEN does *not* include any user data that may follow the last *RF2NVD* field at the end of the structure.

To avoid problems with data conversion of the user data in some environments, consider using *RF2LEN* as a multiple of four.

The following constant gives the length of the *fixed* part of the structure, that is, the length excluding the *RF2NVL* and *RF2NVD* fields:

RFLEN2

Length of fixed part of MQRFH2 structure.

The initial value of this field is RFLEN2.

RF2NVC (10-digit signed integer)

Character set identifier of *RF2NVD*.

This specifies the coded character set identifier of the data in the *RF2NVD* field. This is different from the character set of the other strings in the MQRFH2 structure, and can be different from the character set of the data (if any) that follows the last *RF2NVD* field at the end of the structure.

RF2NVC must have one of the following CCSID values:

1200

UCS-2 open-ended

13488

UCS-2 2.0 subset

17584

UCS-2 2.1 subset (includes the Euro symbol)

1208

UTF-8

For the UCS-2 character sets, the encoding (byte order) of the *RF2NVD* must be the same as the encoding of the other fields in the MQRFH2 structure. Surrogate characters (X'D800' through X'DFFF') are not supported.

Note: If *RF2NVC* does not have one of the values listed previously, and the MQRFH2 structure requires conversion on the MQGET call, the call completes with reason code RC2111 and the message is returned unconverted.

The initial value of this field is 1208.

RF2NVD (n-byte character string)

Name/value data.

This is a variable-length character string containing data encoded using an XML-like syntax. The length in bytes of this string is provided by the *RF2NVL* field that precedes the *RF2NVD* field; this length should be a multiple of four.

The *RF2NVL* and *RF2NVD* fields are optional, but if present they must occur as a pair and be adjacent. The pair of fields can be repeated as many times as required, for example:

```
length1 data1 length2 data2 length3 data3
```

Because these fields are optional, they are omitted from the declarations of the structure that are provided for the various programming languages supported.

RF2NVD is unusual because it is *not* converted to the character set specified on the MQGET call when the message is retrieved with the GMCONV option in effect; *RF2NVD* remains in its original character set. However, *RF2NVD* is converted to the encoding specified on the MQGET call.

Syntax of name/value data: The string consists of a single "folder" that contains zero or more properties. The folder is delimited by XML start and end tags with the same name as the folder:

```
<folder> property1 property2 ... </folder>
```

Characters following the folder end tag, up to the length defined by *RF2NVL*, must be blank. Within the folder, each property is composed of a name and a value, and optionally a data type:

```
<name dt="datatype">value</name>
```

In these examples:

- The delimiter characters (<, =, ", /, and >) must be specified exactly as shown.
- name is the user-specified name of the property; see the following example for more information about names.
- datatype is an optional user-specified data type of the property; see the following example for valid data types.
- value is the user-specified value of the property; see the following paragraphs for more information about values.
- Blanks are significant between the > character which precedes a value, and the < character which follows the value, and at least one blank must precede dt=. Elsewhere blanks can be coded freely between tags, or preceding or following tags (for example, in order to improve readability); these blanks are not significant.

If properties are related to each other, they can be grouped together by enclosing them within XML start and end tags with the same name as the group:

```
<folder> <group> property1 property2 ... </group> </folder>
```

Groups can be nested within other groups, without limit, and a group can occur more than once within a folder. It is also valid for a folder to contain some properties in groups and other properties not in groups.

Names of properties, groups, and folders: Names of properties, groups, and folders must be valid XML tag names, with the exception of the colon character, which is not permitted in a property, group, or folder name. In particular:

- Names must start with a letter or an underscore. Valid letters are defined in the W3C XML specification, and consist essentially of Unicode categories Ll, Lu, Lo, Lt, and Nl.
- The remaining characters in a name can be letters, decimal digits, underscores, hyphens, or dots. These correspond to Unicode categories Ll, Lu, Lo, Lt, Nl, Mc, Mn, Lm, and Nd.
- The Unicode compatibility characters (X'F900' and above) are not permitted in any part of a name.
- Names must not start with the string XML in any mixture of upper or lowercase.

In addition:

- Names are case-sensitive. For example, ABC, abc, and Abc are three different names.
- Each folder has a separate namespace. As a result, a group or property in one folder does not conflict with a group or property of the same name in another folder.
- Groups and properties occupy the same namespace within a folder. As a result, a property cannot have the same name as a group within the folder containing that property.

Generally, programs that analyze the *RF2NVD* field should ignore properties or groups that have names that the program does not recognize, provided that those properties or groups are correctly formed.

Data types of properties: Each property can have an optional data type. If specified, the data type must be one of the following values, in upper, lower, or mixed case:

Data type	Used for
string	Any sequence of characters. Certain characters must be specified using escape sequences.
boolean	The character 0 or 1 (1 denotes TRUE).
bin.hex	Hexadecimal digits representing octets.
i1	Integer number in the range -128 through +127, expressed using only decimal digits and optional sign.
i2	Integer number in the range -32 768 through +32 767, expressed using only decimal digits and optional sign.
i4	Integer number in the range -2 147 483 648 through +2 147 483 647, expressed using only decimal digits and optional sign.
i8	Integer number in the range -9 223 372 036 854 775 808 through +9 223 372 036 854 775 807, expressed using only decimal digits and optional sign.
int	Integer number in the range -9 223 372 036 854 775 808 through +9 223 372 036 854 775 807, expressed using only decimal digits and optional sign. This can be used in place of i1, i2, i4, or i8 if the sender does not want to imply a particular precision.
r4	Floating-point number with magnitude in the range 1.175E-37 through 3.402 823 47E+38, expressed using decimal digits, optional sign, optional fractional digits, and optional exponent.
r8	Floating-point number with magnitude in the range 2.225E-307 through 1.797 693 134 862 3E+308 expressed using decimal digits, optional sign, optional fractional digits, and optional exponent.

Values of properties: The value of a property can consist of any characters, except as detailed in the following figure. Each occurrence in the value of a character marked as "mandatory" must be replaced by the corresponding escape sequence. Each occurrence in the value of a character marked as "optional" can be replaced by the corresponding escape sequence, but this is not required.

Character	Escape sequence	Usage
&	&	Mandatory
<	<	Mandatory
>	>	Optional
"	"	Optional
'	'	Optional

Note: The & character at the start of an escape sequence must *not* be replaced by &;.

In the following example, the blanks in the value are significant; however, no escape sequences are needed:

```
<Famous_Words>The program displayed "Hello World"</Famous_Words>
```

RF2NVL (10-digit signed integer)

Length of *RF2NVD*.

This specifies the length in bytes of the data in the *RF2NVD* field. To avoid problems with data conversion of the data (if any) that *follows* the *RF2NVD* field, *RF2NVL* should be a multiple of four.

Note: The *RF2NVL* and *RF2NVD* fields are optional, but if present they must occur as a pair and be adjacent. The pair of fields can be repeated as many times as required, for example:

```
length1 data1 length2 data2 length3 data3
```

Because these fields are optional, they are omitted from the declarations of the structure that are provided for the various programming languages supported.

RF2SID (4-byte character string)

Structure identifier.

The value must be:

RFSIDV

Identifier for rules and formatting header structure.

The initial value of this field is RFSIDV.

RF2VER (10-digit signed integer)

Structure version number.

The value must be:

RFVER2

Version-2 rules and formatting header structure.

The initial value of this field is RFVER2.

Initial values

Table 609. Initial values of fields in MQRFH2		
Field name	Name of constant	Value of constant
<i>RF2SID</i>	RFSIDV	'RFH¬'
<i>RF2VER</i>	RFVER2	2
<i>RF2LEN</i>	RLEN2	36
<i>RF2ENC</i>	ENNAT	Depends on environment
<i>RF2CSI</i>	CSINHT	-2
<i>RF2FMT</i>	FMNONE	Blanks
<i>RF2FLG</i>	RFNONE	0
<i>RF2NVC</i>	None	1208
Notes:		
1. The symbol ¬ represents a single blank character.		

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D*  
D* MQRFH2 Structure
```

```

D*
D* Structure identifier
D  RF2SID          1          4      INZ('RFH ')
D* Structure version number
D  RF2VER          5          8I 0 INZ(2)
D* Total length of MQRFH2 including allNameValueLength and
D* NameValueDatafields
D  RF2LEN          9          12I 0 INZ(36)
D* Numeric encoding of data that follows last NameValueData field
D  RF2ENC          13         16I 0 INZ(273)
D* Character set identifier of data that follows last NameValueData field
D  RF2CSI          17         20I 0 INZ(-2)
D* Format name of data that follows last NameValueData field
D  RF2FMT          21         28      INZ(' ')
D* Flags
D  RF2FLG          29         32I 0 INZ(0)
D* Character set identifier of NameValueData
D  RF2NVC          33         36I 0 INZ(1208)

```

MQRMH - Reference message header

The MQRMH structure defines the format of a reference message header.

Overview

Purpose: This header is used with user-written message channel exits to send large amounts of data (called "bulk data") from one queue manager to another. The difference compared to normal messaging is that the bulk data is not stored on a queue; instead, only a *reference* to the bulk data is stored on the queue. This reduces the possibility of IBM MQ resources being exhausted by a few large messages.

Format name: FMRMH.

Character set and encoding: Character data in MQRMH, and the strings addressed by the offset fields, must be in the character set of the local queue manager; this is given by the *CodedCharSetId* queue manager attribute. Numeric data in MQRMH must be in the native machine encoding; this is given by the value of ENNAT for the C programming language.

The character set and encoding of the MQRMH must be set into the *MDCSI* and *MDENC* fields in:

- The MQMD (if the MQRMH structure is at the start of the message data), or
- The header structure that precedes the MQRMH structure (all other cases).

Usage: An application puts a message consisting of an MQRMH, but omitting the bulk data. When the message is read from the transmission queue by a message channel agent (MCA), a user-supplied message exit is invoked to process the reference message header. The exit can append to the reference message the bulk data identified by the MQRMH structure, before the MCA sends the message through the channel to the next queue manager.

At the receiving end, a message exit that waits for reference messages should exist. When a reference message is received, the exit should create the object from the bulk data that follows the MQRMH in the message, and then pass on the reference message without the bulk data. The reference message can later be retrieved by an application reading the reference message (without the bulk data) from a queue.

Normally, the MQRMH structure is all that is in the message. However, if the message is on a transmission queue, one or more additional headers will precede the MQRMH structure.

A reference message can also be sent to a distribution list. In this case, the MQDH structure and its related records precede the MQRMH structure when the message is on a transmission queue.

Note: A reference message should not be sent as a segmented message, because the message exit cannot process it correctly.

- [“Data conversion” on page 1158](#)
- [“Fields” on page 1158](#)
- [“Initial values” on page 1162](#)
- [“RPG declaration” on page 1163](#)

Data conversion

For data conversion purposes, conversion of the MQRMH structure includes conversion of the source environment data, source object name, destination environment data, and destination object name. Any other bytes within *RMLLEN* bytes of the start of the structure are either discarded or have undefined values after data conversion. The bulk data will be converted provided that all of the following are true:

- The bulk data is present in the message when the data conversion is performed.
- The *RMFMT* field in MQRMH has a value other than FMNONE.
- A user-written data-conversion exit exists with the format name specified.

Be aware, however, that typically the bulk data is *not* present in the message when the message is on a queue, and that as a result the bulk data will not be converted by the GMCONV option.

Fields

The MQRMH structure contains the following fields; the fields are described in **alphabetical order**:

RMCSI (10-digit signed integer)

Character set identifier of bulk data.

This specifies the character set identifier of the bulk data; it does not apply to character data in the MQRMH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

RMDEL (10-digit signed integer)

Length of destination environment data.

If this field is zero, there is no destination environment data, and *RMDEO* is ignored.

RMDEO (10-digit signed integer)

Offset of destination environment data.

This field specifies the offset of the destination environment data from the start of the MQRMH structure. Destination environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, the destination environment data might be the directory path of the object where the bulk data is to be stored. However, if the creator does not know the destination environment data, it is the responsibility of the user-supplied message exit to determine any environment information needed.

The length of the destination environment data is given by *RMDEL* ; if this length is zero, there is no destination environment data, and *RMDEO* is ignored. If present, the destination environment data must reside completely within *RMLLEN* bytes from the start of the structure.

Applications should not assume that the destination environment data is contiguous with any of the data addressed by the *RMSEO*, *RMSNO*, and *RMDNO* fields.

The initial value of this field is 0.

RMDL (10-digit signed integer)

Length of bulk data.

The *RMDL* field specifies the length of the bulk data referenced by the MQRMH structure.

If the bulk data is present in the message, the data begins at an offset of *RMLen* bytes from the start of the MQRMH structure. The length of the entire message minus *RMLen* gives the length of the bulk data present.

If data is present in the message, *RMDL* specifies the amount of that data that is relevant. The normal case is for *RMDL* to have the same value as the length of data present in the message.

If the MQRMH structure represents the remaining data in the object (starting from the specified logical offset), the value zero can be used for *RMDL*, if the bulk data is not present in the message.

If no data is present, the end of MQRMH coincides with the end of the message.

The initial value of this field is 0.

RMDNL (10-digit signed integer)

Length of destination object name.

If this field is zero, there is no destination object name, and *RMDNO* is ignored.

RMDNO (10-digit signed integer)

Offset of destination object name.

This field specifies the offset of the destination object name from the start of the MQRMH structure. The destination object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the destination object name, it is the responsibility of the user-supplied message exit to identify the object to be created or modified.

The length of the destination object name is given by *RMDNL* ; if this length is zero, there is no destination object name, and *RMDNO* is ignored. If present, the destination object name must reside completely within *RMLen* bytes from the start of the structure.

Applications should not assume that the destination object name is contiguous with any of the data addressed by the *RMSEO*, *RMSNO*, and *RMDEO* fields.

The initial value of this field is 0.

RMDO (10-digit signed integer)

Low offset of bulk data.

This field specifies the low offset of the bulk data from the start of the object of which the bulk data forms part. The offset of the bulk data from the start of the object is called the *logical offset*. This is *not* the physical offset of the bulk data from the start of the MQRMH structure - that offset is given by *RMLen*.

To allow large objects to be sent using reference messages, the logical offset is divided into two fields, and the actual logical offset is given by the sum of these two fields:

- *RMDO* represents the remainder obtained when the logical offset is divided by 1 000 000 000. It is thus a value in the range 0 through 999 999 999.
- *RMDO2* represents the result obtained when the logical offset is divided by 1 000 000 000. It is thus the number of complete multiples of 1 000 000 000 that exist in the logical offset. The number of multiples is in the range 0 through 999 999 999.

The initial value of this field is 0.

RMDO2 (10-digit signed integer)

High offset of bulk data.

This field specifies the high offset of the bulk data from the start of the object of which the bulk data forms part. It is a value in the range 0 through 999 999 999. See *RMDO* for details.

The initial value of this field is 0.

RMENC (10-digit signed integer)

Numeric encoding of bulk data.

This specifies the numeric encoding of the bulk data; it does not apply to numeric data in the MQRMH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is ENNAT.

RMFLG (10-digit signed integer)

Reference message flags.

The following flags are defined:

RMLAST

Reference message contains or represents last part of object.

This flag indicates that the reference message represents or contains the last part of the referenced object.

RMNLST

Reference message does not contain or represent last part of object.

RMNLST is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is RMNLST.

RMFMT (8-byte character string)

Format name of bulk data.

This specifies the format name of the bulk data.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

The initial value of this field is FMNONE.

RMLEN (10-digit signed integer)

Total length of MQRMH, including strings at end of fixed fields, but not the bulk data.

The initial value of this field is zero.

RMOII (24-byte bit string)

Object instance identifier.

This field can be used to identify a specific instance of an object. If it is not needed, it should be set to the following value:

OIINON

No object instance identifier specified.

The value is binary zero for the length of the field.

The length of this field is given by LNOIID. The initial value of this field is OIINON.

RMOT (8-byte character string)

Object type.

This is a name that can be used by the message exit to recognize types of reference message that it supports. Consider making the name conform to the same rules as the *RMFMT* field.

The initial value of this field is 8 blanks.

RMSEL (10-digit signed integer)

Length of source environment data.

If this field is zero, there is no source environment data, and *RMSEO* is ignored.

The initial value of this field is 0.

RMSEO (10-digit signed integer)

Offset of source environment data.

This field specifies the offset of the source environment data from the start of the MQRMH structure. Source environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, the source environment data might be the directory path of the object containing the bulk data. However, if the creator does not know the source environment data, it is the responsibility of the user-supplied message exit to determine any environment information needed.

The length of the source environment data is given by *RMSEL* ; if this length is zero, there is no source environment data, and *RMSEO* is ignored. If present, the source environment data must reside completely within *RMLLEN* bytes from the start of the structure.

Applications should not assume that the environment data starts immediately after the last fixed field in the structure or that it is contiguous with any of the data addressed by the *RMSNO*, *RMDEO*, and *RMDNO* fields.

The initial value of this field is 0.

RMSID (4-byte character string)

Structure identifier.

The value must be:

RMSIDV

Identifier for reference message header structure.

The initial value of this field is RMSIDV.

RMSNL (10-digit signed integer)

Length of source object name.

If this field is zero, there is no source object name, and *RMSNO* is ignored.

The initial value of this field is 0.

RMSNO (10-digit signed integer)

Offset of source object name.

This field specifies the offset of the source object name from the start of the MQRMH structure. The source object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the source object name, it is the responsibility of the user-supplied message exit to identify the object to be accessed.

The length of the source object name is given by *RMSNL* ; if this length is zero, there is no source object name, and *RMSNO* is ignored. If present, the source object name must reside completely within *RMLLEN* bytes from the start of the structure.

Applications should not assume that the source object name is contiguous with any of the data addressed by the *RMSEO*, *RMDEO*, and *RMDNO* fields.

The initial value of this field is 0.

RMVER (10-digit signed integer)

Structure version number.

The value must be:

RMVER1

Version-1 reference message header structure.

The following constant specifies the version number of the current version:

RMVERC

Current version of reference message header structure.

The initial value of this field is RMVER1.

Initial values

Table 610. Initial values of fields in MQRMH		
Field name	Name of constant	Value of constant
<i>RMSID</i>	RMSIDV	'RMH¬ '
<i>RMVER</i>	RMVER1	1
<i>RMLLEN</i>	None	0
<i>RMENC</i>	ENNAT	Depends on environment
<i>RMCSI</i>	CSUNDF	0
<i>RMFMT</i>	FMNONE	Blanks
<i>RMFLG</i>	RMNLST	0
<i>RMOT</i>	None	Blanks
<i>RMOII</i>	OIINON	Nulls
<i>RMSEL</i>	None	0
<i>RMSEO</i>	None	0
<i>RMSNL</i>	None	0
<i>RMSNO</i>	None	0
<i>RMDEL</i>	None	0
<i>RMDEO</i>	None	0
<i>RMDNL</i>	None	0
<i>RMDNO</i>	None	0
<i>RMDL</i>	None	0
<i>RMDO</i>	None	0
<i>RMDO2</i>	None	0
Notes:		
1. The symbol ¬ represents a single blank character.		

```

D*.1.....2.....3.....4.....5.....6.....7..
D*
D* MQRMH Structure
D*
D* Structure identifier
D RMSID          1      4      INZ('RMH ')
D* Structure version number
D RMVER          5      8I 0 INZ(1)
D* Total length of MQRMH, including strings at end of fixed fields, but not

```

```

D* the bulk data
D  RMLEN          9      12I 0 INZ(0)
D* Numeric encoding of bulk data
D  RMENC         13      16I 0 INZ(273)
D* Character set identifier of bulkdata
D  RMCSI         17      20I 0 INZ(0)
D* Format name of bulk data
D  RMFMT         21      28      INZ('      ')
D* Reference message flags
D  RMFLG         29      32I 0 INZ(0)
D* Object type
D  RMOT          33      40      INZ
D* Object instance identifier
D  RMOII         41      64      INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D* Length of source environmentdata
D  RMSEL         65      68I 0 INZ(0)
D* Offset of source environmentdata
D  RMSEO         69      72I 0 INZ(0)
D* Length of source object name
D  RMSNL         73      76I 0 INZ(0)
D* Offset of source object name
D  RMSNO         77      80I 0 INZ(0)
D* Length of destination environmentdata
D  RMDEL         81      84I 0 INZ(0)
D* Offset of destination environmentdata
D  RMDEO         85      88I 0 INZ(0)
D* Length of destination objectname
D  RMDNL         89      92I 0 INZ(0)
D* Offset of destination objectname
D  RMDNO         93      96I 0 INZ(0)
D* Length of bulk data
D  RMDL          97     100I 0 INZ(0)
D* Low offset of bulk data
D  RMDO         101     104I 0 INZ(0)
D* High offset of bulk data
D  RMDO2        105     108I 0 INZ(0)

```

RPG declaration

MQRR - Response record

The MQRR structure is used to receive the completion code and reason code resulting from the open or put operation for a single destination queue, when the destination is a distribution list.

Overview

Purpose: MQRR is an output structure for the MQOPEN, MQPUT, and MQPUT1 calls.

Character set and encoding: Data in MQRR must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

Usage: By providing an array of these structures on the MQOPEN and MQPUT calls, or on the MQPUT1 call, it is possible to determine the completion codes and reason codes for all of the queues in a distribution list when the outcome of the call is mixed, that is, when the call succeeds for some queues in the list but fails for others. Reason code RC2136 from the call indicates that the response records (if provided by the application) have been set by the queue manager.

- [“Fields” on page 1163](#)
- [“Initial values” on page 1164](#)
- [“RPG declaration” on page 1164](#)

Fields

The MQRR structure contains the following fields; the fields are described in **alphabetical order**:

RRCC (10-digit signed integer)

Completion code for queue.

This is the completion code resulting from the open or put operation for the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is CCOK.

RRREA (10-digit signed integer)

Reason code for queue.

This is the reason code resulting from the open or put operation for the queue with the name that was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is RCNONE.

Initial values

Table 611. Initial values of fields in MQRR		
Field name	Name of constant	Value of constant
RRCC	CCOK	0
RRREA	RCNONE	0

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D*  
D* MQRR Structure  
D*  
D* Completion code for queue  
D  RRCC              1      4I 0 INZ(0)  
D* Reason code for queue  
D  RRREA             5      8I 0 INZ(0)
```

MQSCO - SSL configuration options

The MQSCO structure (with the SSL fields in the MQCD structure) allows an application running as an IBM MQ MQI client to specify configuration options that control the use of SSL for the client connection when the channel protocol is TCP/IP.

Overview

Purpose: The structure is an input parameter on the MQCONN call.

If the channel protocol for the client channel is not TCP/IP, the MQSCO structure is ignored.

Character set and encoding: Data in MQSCO must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT.

- [“Fields” on page 1164](#)
- [“Initial values” on page 1168](#)
- [“RPG declaration” on page 1168](#)

Fields

The MQSCO structure contains the following fields; the fields are described in **alphabetical order**:

SCAIC (10-digit signed integer)

This is the number of authentication information (MQAIR) records addressed by the *SCAIP* or *SCAIO* fields. For more information, see “MQAIR - Authentication information record” on page 976. The value must be zero or greater. If the value is not valid, the call fails with reason code RC2383.

This is an input field. The initial value of this field is 0.

SCAIO (10-digit signed integer)

This is the offset in bytes of the first authentication information record from the start of the MQSCO structure. The offset can be positive or negative. The field is ignored if *SCAIC* is zero.

You can use either *SCAIO* or *SCAIP* to specify the MQAIR records, but not both; see the description of the *SCAIP* field for details.

This is an input field. The initial value of this field is 0.

SCAIP (10-digit signed integer)

This is the address of the first authentication information record. The field is ignored if *SCAIC* is zero.

You can provide the array of MQAIR records in one of two ways:

- By using the pointer field *SCAIP*

In this case, the application can declare an array of MQAIR records that is separate from the MQSCO structure, and set *SCAIP* to the address of the array.

Consider using *SCAIP* for programming languages that support the pointer data type in a fashion that is portable to different environments (for example, the C programming language).

- By using the offset field *SCAIO*

In this case, the application must declare a compound structure containing an MQSCO followed by the array of MQAIR records, and set *SCAIO* to the offset of the first record in the array from the start of the MQSCO structure. Ensure that this value is correct, and has a value that can be accommodated within an MQLONG (the most restrictive programming language is COBOL, for which the valid range is -999 999 999 through +999 999 999).

Consider using *SCAIO* for programming languages that do not support the pointer data type, or that implement the pointer data type in a fashion that is not portable to different environments (for example, the COBOL programming language).

Whatever technique you choose, only one of *SCAIP* and *SCAIO* can be used; the call fails with reason code RC2384 if both are nonzero.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

SCCERLBL (10-digit signed integer)

This field gives details of the certificate label being used.

IBM MQ initializes the value for the SCCERLBL field as blanks. Either enter the required value, or accept the default value.

`ibmwebsphermq<user_id>` is a valid value for this field for all versions of the product, and for MQSCO versions less than 5.0 it is the only valid value. Therefore the value of this field is interpreted at run time, and changed if necessary. If you specify an MQSCO version less than 5.0, or accept the default value of blanks for the SCCERLBL field, the system uses the value `ibmwebsphermq<user_id>`.

This is an input field.

SCCERTVPOL (10-digit signed integer)

This field specifies what type of certificate validation policy is used. The field can be set to one of the following values:

MQ_CERT_VAL_POLICY_ANY

Apply each of the certificate validation policies supported by the secure sockets library. Accept the certificate chain if any of the policies considers the certificate chain valid.

MQ_CERT_VAL_POLICY_RFC5280

Apply only the RFC5280 compliant certificate validation policy. This setting provides stricter validation than the ANY setting, but rejects some older digital certificates.

The initial value of this field is MQ_CERT_VAL_POLICY_ANY

SCCH (10-digit signed integer)

This field gives configuration details for cryptographic hardware connected to the client system.

Set the field to a string in the following format, or leave it blank or null:

```
GSK_PKCS11=<the PKCS #11 driver path and file name>;<the PKCS #11 token label>;<the PKCS #11 token password>;<symmetric cipher setting>;
```

To use cryptographic hardware which conforms to the PKCS11 interface, for example, the IBM 4960 or IBM 4963, specify the PKCS11 driver path, PKCS11 token label, and PKCS11 token password strings, each terminated by a semi-colon.

The PKCS #11 driver path is an absolute path to the shared library providing support for the PKCS #11 card. The PKCS #11 driver file name is the name of the shared library. An example of the value required for the PKCS #11 path and file name is:

```
/usr/lib/pkcs11/PKCS11_API.so
```

The PKCS #11 token label must be entirely in lowercase. If you have configured your hardware with a mixed case or uppercase token label, reconfigure it with this lowercase label.

If no cryptographic hardware configuration is required, set the field to blank or null.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. If the value is not valid, or leads to a failure when used to configure the cryptographic hardware, the call fails with reason code RC2382.

This is an input field. The length of this field is given by LNSSCH. The initial value of this field is blank characters.

SCEPSUITEB (10-digit signed integer)

This field Specifies whether Suite B compliant cryptography is used and what level of strength is employed. The value can be one or more of:

- SCEPSUITEB0

Suite B compliant cryptography is not used.

- SCEPSUITEB1

Suite B 128-bit strength security is used.

- SCEPSUITEB2

Suite B 192-bit strength security is used.

Note: Using SCEPSUITEB0 with any other value in this field is invalid.

SCFR (10-digit signed integer)

IBM MQ can be configured with cryptographic hardware so that the cryptography modules used are those provided by the hardware product; these can be FIPS-certified to a particular level depending on the cryptographic hardware product in use.

Use this field to specify that only FIPS-certified algorithms are used if the cryptography is provided in IBM MQ-provided software.

When IBM MQ is installed an implementation of SSL cryptography is also installed which provides some FIPS-certified modules.

The values can be:

MQSSL_FIPS_NO

This is the default value. When set to this value:

- Any CipherSpec supported on a particular platform can be used.
- If run without use of cryptographic hardware, the following CipherSpecs run using FIPS 140-2 certified cryptography on the IBM MQ platforms:
 - TLS_RSA_WITH_3DES_EDE_CBC_SHA
 - TLS_RSA_WITH_AES_128_CBC_SHA
 - TLS_RSA_WITH_AES_256_CBC_SHA

MQSSL_FIPS_YES

When set to this value, unless you are using cryptographic hardware to perform the cryptography, you can be sure that

- Only FIPS-certified cryptographic algorithms can be used in the CipherSpec applying to this client connection.
- Inbound and outbound SSL channel connections only succeed if one of the following Cipher Specs are used:
 - TLS_RSA_WITH_3DES_EDE_CBC_SHA
 - TLS_RSA_WITH_AES_128_CBC_SHA
 - TLS_RSA_WITH_AES_256_CBC_SHA

Notes:

1. CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated.
2. Where possible, if FIPS-only CipherSpecs is configured then the MQI client rejects connections which specify a non-FIPS CipherSpec with RC2393. IBM MQ does not guarantee to reject all such connections and it is your responsibility to determine whether your IBM MQ configuration is FIPS-compliant.

SCKR (10-digit signed integer)

This field is relevant only for IBM MQ MQI clients running on UNIX systems and Windows systems. It specifies the location of the key database file in which keys and certificates are stored. The key database file must have a file name of the form `zzz.kdb`, where `zzz` is user-selectable. The *SCKR* field contains the path to this file, along with the file name stem (all characters in the file name up to but not including the final `.kdb`). The `.kdb` file suffix is added automatically.

Each key database file has an associated *password stash file*. This holds encrypted passwords that are used to allow programmatic access to the key database. The password stash file must reside in the same directory and have the same file stem as the key database, and must end with the suffix `.sth`.

For example, if the *SCKR* field has the value `/xxx/yyy/key`, the key database file must be `/xxx/yyy/key.kdb`, and the password stash file must be `/xxx/yyy/key.sth`, where `xxx` and `yyy` represent directory names.

If the value is shorter than the length of the field, terminate the value with a null character, or pad it with blanks to the length of the field. The value is not checked; if there is an error in accessing the key repository, the call fails with reason code RC2381.

To run an SSL connection from an IBM MQ MQI client, set *SCKR* to a valid key database file name.

This is an input field. The length of this field is given by LNSSKR. The initial value of this field is a blank character.

SCSID (10-digit signed integer)

This is the structure identifier; the value must be:

SCSIDV

Identifier for SSL configuration options structure.

This is always an input field. The initial value of this field is SCSIDV.

SCVER (10-digit signed integer)

This is the structure version number; the value must be:

SCVER1

Version-1 SSL configuration options structure.

SCVER2

Version-2 SSL configuration options structure.

The following constant specifies the version number of the current version:

SCVERC

Current version of SSL configuration options structure.

This is always an input field. The initial value of this field is SCVER2

Initial values

Table 612. Initial values of fields in MQSCO		
Field name	Name of constant	Value of constant
SCSID	SCSIDV	'SCO~'
SCVER	SCVER5	1
SCKR	None	Null string or blanks
SCCH	None	Null string or blanks
SCAIC	None	0
SCAIO	None	0
SCAIP	None	Null pointer or null bytes
SCKRC	None	Null pointer or null bytes
SCFR	None	Null pointer or null bytes
SCEPSUITEB	None	Null pointer or null bytes
SCCERTVPOL	None	Null pointer or null bytes
SCCERLBL	None	Null pointer or null bytes
Notes:		
1. The symbol ~ represents a single blank character.		
2. See “RPG declaration” on page 1168 for the SCEPSUITEB options.		

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D* MQSCO Structure  
D*  
D* Structure identifier  
D  SCSID          1      4    INZ('SCO ')  
D* Structure version number  
D  SCVER          5      8I 0 INZ(1)
```

```

D* Location of SSL key repository
D SCKR          9      264      INZ
D* Cryptographic hardware configuration string
D SCCH          265      520      INZ
D* Number of MQAIR records present
D SCAIC          521      524I 0 INZ(0)
D* Offset of first MQAIR record from start of MQSCO structure
D SCAIO          525      528I 0 INZ(0)
D* Address of first MQAIR record
D SCAIP          529      544*      INZ(*NULL)
D* Ver:1 **
D* Number of unencrypted bytes sent/received before secret key is
D* reset
D SCKRC          545      548I 0 INZ(0)
D* Using FIPS-certified algorithms
D SCFR          549      552I 0 INZ(0)
D* Ver:2 **
* Use only Suite B cryptographic algorithms
D SCEPSUITEB0
D SCEPSUITEB1          553      556I 0 INZ(1)
D SCEPSUITEB2          557      560I 0 INZ(0)
D SCEPSUITEB3          561      564I 0 INZ(0)
D SCEPSUITEB4          565      568I 0 INZ(0)
D SCEPSUITEB          10I 0 DIM(4) OVERLAY(SCEPSUITEB0)
D* Ver:3 **
D* Certificate validation policy
D SCCERTVPOL          569      572I 0 INZ(0)
D* Ver:4 **

```

MQSD - Subscription descriptor

The MQSD structure is used to specify details about the subscription being made.

Overview

Purpose

The structure is an input/output parameter on the MQSUB call.

Managed subscriptions

If an application has no specific need to use a particular queue as the destination for those publications that match its subscription, it can use the managed subscription feature. If an application elects to use a managed subscription, the queue manager informs the subscriber about the destination where published messages are sent, by providing an object handle as an output from the MQSUB call. For more information, see [HOBJ \(10-digit signed integer\) - input/output](#).

When the subscription is removed, the queue manager also undertakes to clean up messages that have not been retrieved from the managed destination, in the following situations:

- When the subscription is removed - by use of MQCLOSE with CORMSB - and the managed Hobj is closed.
- By implicit means when the connection is lost to an application using a non-durable subscription (SONDUR)
- By expiration when a subscription is removed because it has expired and the managed Hobj is closed.

You must use managed subscriptions with non-durable subscriptions, so that the clean up can occur, and so that messages for closed non-durable subscriptions do not take up space in your queue manager. Durable subscriptions can also use managed destinations.

Character set and encoding

Data in MQSD must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT. However, if the application is running as an IBM MQ client, the structure must be in the character set and encoding of the client.

- [“Fields” on page 1170](#)
- [“Initial values” on page 1181](#)
- [“RPG declaration” on page 1182](#)

Fields

The MQSD structure contains the following fields; the fields are described in alphabetical order:

SDAID (32 byte character string)

This value is in the *MDAID* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. *SDAID* is part of the identity context of the message. For more information about message context, see [Message context](#).

For more information about *MDAID* see [MDAID](#).

If the *S0SETI* option is not specified, the *MDAID* which is set in each message published for this subscription is blanks, as default context information.

If the *S0SETI* option is specified, the *SDAID* is being generated by the user and this field is an input field which contains the *MDAID* to be set in each publication for this subscription.

The length of this field is given by *LNAIDD*. The initial value of this field is 32 blank characters.

If altering an existing subscription using the *SOALT* option, the *SDAID* of any future publication messages can be changed.

On return from an *MQSUB* call using *SORES*, this field is set to the current *MDAID* being used for the subscription.

SDACC (32 byte character string)

This value is in the *MDACC* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. *MDACC* is part of the identity context of the message. For more information about message context, see [Message context](#).

For more information about *MDACC* see [MDACC](#).

You can use the following special value for the *SDACC* field:

ACNONE

No accounting token is specified.

The value is binary zero for the length of the field.

If the *S0SETI* option is not specified, the accounting token is generated by the queue manager as default context information and this field is an output field which contains the *MDACC* which is set in each message published for this subscription.

If the *S0SETI* option is specified, the accounting token is being generated by the user and this field is an input field which contains the *MDACC* to be set in each publication for this subscription.

The length of this field is given by *LNACCT*. The initial value of this field is *ACNONE*.

If altering an existing subscription using the *SOALT* option, the value of *MDACC* in any future publication messages can be changed.

On return from an *MQSUB* call using *SORES*, this field is set to the current *MDACC* being used for the subscription.

SDASI (40 byte bit string)

This is a security identifier that is passed with the *SDAU* to the authorization service to allow appropriate authorization checks to be performed.

SDASI is used only if *SOALTU* is specified, and the *SDAU* field is not entirely blank up to the first null character or the end of the field.

On return from an *MQSUB* call using *SORES*, this field is unchanged.

See the description of [ODASI](#) in the MQOD data type for more information.

SDAU (12 byte character string)

If you specify SOALTU, this field contains an alternate user identifier that is used to check the authorization for the subscription and for output to the destination queue (specified in the *Hobj* parameter of the MQSUB call), in place of the user identifier that the application is currently running under.

If successful, the user identifier specified in this field is recorded as the subscription owning user identifier in place of the user identifier that the application is currently running under.

If SOALTU is specified and this field is entirely blank up to the first null character or the end of the field, the subscription can succeed only if no user authorization is needed to subscribe to this topic with the options specified or the destination queue for output.

If SOALTU is not specified, this field is ignored.

On return from an MQSUB call using SORES, this field is unchanged.

This is an input field. The length of this field is given by LNUID. The initial value of this field is 12 blank characters.

SDCID (24 byte bit string)

All publications sent to match this subscription contain this correlation identifier in the message descriptor. If multiple subscriptions use the same queue to get their publications from, using MQGET by correlation ID allows only publications for a specific subscription to be obtained. This correlation identifier can either be generated by the queue manager or by the user.

If the S0SCID option is not specified, the correlation identifier is generated by the queue manager and this field is an output field which contains the correlation identifier which is set in each message published for this subscription.

If the S0SCID option is specified, the correlation identifier is being generated by the user and this field is an input field which contains the correlation identifier to be set in each publication for this subscription. In this case, if the field contains CINONE, the correlation identifier which is set in each message published for this subscription is the correlation identifier that was created by the original put of the message.

If the S0GRP option is specified and the correlation identifier specified is the same as an existing grouped subscription using the same queue and an overlapping topic string, only the most significant subscription in the group is provided with a copy of the publication.

The length of this field is given by LNCID. The initial value of this field is CINONE.

If altering an existing subscription using the SOALT option, and this field is an input field, then the subscription correlation ID can be changed, unless the subscription has been created using the S0GRP option.

On return from an MQSUB call using SORES, this field is set to the current correlation ID for the subscription.

SDEXP (10 digit signed integer)

This is the time expressed in tenths of a second after which the subscription expires. No more publications will match this subscription after this interval has passed. This is also used as the value in the *MDEXP* field in the MQMD of the publications sent to this subscriber.

The following special value is recognized:

EIULIM

The subscription has an unlimited expiration time.

If altering an existing subscription using the SOALT option, the expiry of the subscription can be changed.

On return from an MQSUB call using the SORES option this field is set to the original expiry of the subscription and not the remaining expiry time.

SDON (48 byte character string)

This is the name of the topic object as defined on the local queue manager.

The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but can contain trailing blanks. Use a null character to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- Names containing lowercase characters, forward slash, or percent, must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified for names that occur as fields in structures or as parameters on calls.

The *SDON* is used to form the Full topic name.

The full topic name can be built from two different fields: *SDON* and *SDOS*. For details of how these two fields are used, see [“Using topic strings” on page 554](#).

On return from an MQSUB call using the SORES option this field is unchanged.

The length of this field is given by LNTOPN. The initial value of this field is 48 blank characters.

If altering an existing subscription using the SDALT option, the name of the topic object subscribed to cannot be changed. This field and *SDOS* can be omitted. If they are provided they must resolve to the same full topic name or the call fails with RC2510 .

SDOPT (10 digit signed integer)

You must specify at least one of the following options:

- SOALT
- SORES
- SOCRT

The values can be added. Do not add the same constant more than once. The table shows how you can combine these options: Combinations that are not valid are noted; any other combinations are valid.

Access or creation options

Access and creation options control whether a subscription is created, or whether an existing subscription is returned or altered. You must specify at least one of these options. The table displays valid combinations of access or creation options.

Combination of options	Notes
SOCRT	Creates a subscription if one does not exist; fails if the subscription exists.
SORES	Resumes an existing subscription, fails if no subscription exists.
SOCRT + SORES	Creates a subscription if one does not exist and resumes a matching one, if it does exist. Useful combination if used in an application that might be run a number of times.

Combination of options	Notes
SORES + SOALT (see note)	Resumes an existing subscription, altering any fields to match those specified in the MQSD, fails if no subscription exists.
SOCRT + SOALT (see note)	Creates a subscription if one does not exist and resumes a matching one, if it does exist, altering any fields to match those specified in the MQSD. Useful combination if used in an application that wants to ensure that its subscription is in a certain state before proceeding.

Note:

Options specifying SOALT can also specify SORES, but this combination has no additional effect to specifying SOALT alone. SOALT implies SORES, because calling MQSUB to alter a subscription implies that the subscriptions are also resumed. The opposite is not true, however: resuming a subscription does not imply it is to be altered.

SOCRT

Create a subscription for the topic specified. If a subscription using the same *SDSN* exists, the call fails with RC2432 . This failure can be avoided by combining the SOCRT option with SORES. The *SDSN* is not always necessary. For more details, see the description of that field.

Combining SOCRT with SORES first checks whether there is an existing subscription for the specified *SDSN*, and if there is returns a handle to that preexisting subscription; but if there is no existing subscription, a new one is created using all the fields provided in the MQSD.

SOCRT can also be combined with SOALT to similar effect (see details about SOALT later in this topic).

SORES

Return a handle to a preexisting subscription which matches those specified by *SDSN*. No changes are made to the matching subscription attributes, and they are returned on output in the MQSD structure. Most of the contents of the MQSD are not used: The fields used are *SDSID*, *SDVER*, *SDOPT*, *SDAID* and *SDASI*, and *SDSN*.

The call fails with reason code RC2428 if a subscription does not exist matching the full subscription name. This failure can be avoided by combining the SOCRT option with SORES. For details about SOCRT, see [SOCRT](#).

The user ID of the subscription is the user ID that created the subscription, or if it has been later altered by a different user ID, it is the user ID of the most recent, successful alteration. If an *SDAID* is used, and use of alternate user IDs is allowed for that user, *SDAID* is recorded as the user ID that created the subscription instead of the user ID under which the subscription was made.

The user ID that created the subscription is recorded as *SDAU* if that field is used, and the use of alternate user IDs is allowed for that user.

If a matching subscription exists which was created without the SOAUID option and the user ID of the subscription is different from that of the application requesting a handle to the subscription, the call fails with reason code RC2434 .

If a matching subscription exists and is currently in use by another application, the call fails with reason code RC2429 . If it is currently in use by the same connection, the call does not fail and a handle to the subscription is returned.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call fails with RC2523 .

SORES is implied by SOALT and so is not required to be combined with that option, however, it is not an error if those two options are combined.

SOALT

Return a handle to a preexisting subscription with the full subscription name matching those specified in *SDSN*. Any attributes of the subscription that are different from those specified in the MQSD is altered in the subscription unless alteration is disallowed for that attribute. Details are noted in the description of each attribute and are summarized in the following table. If you try to alter an attribute that cannot be changed, the call fails with the reason code shown in the following table.

The call fails with reason code RC2428 if a subscription does not exist matching the full subscription name. This failure can be avoided by combining the SOCRT option with SOALT.

Combining SOCRT with SOALT first checks whether there is an existing subscription for the specified full subscription name, and if there is returns a handle to that preexisting subscription with alterations made as previously detailed; but if there is no existing subscription, a new one is created using all the fields provided in the MQSD.

The user ID of the subscription is the user ID that created the subscription, or if it has been later altered by a different user ID, it is the user ID of the most recent successful alteration. If *SDAU* is used (and use of alternate user IDs is allowed for that user), then the alternate user ID is recorded as the user ID that created the subscription instead of the user ID under which the subscription was made.

If a matching subscription exists that was created without the option SOAUID and the user ID of the subscription is different from that of the application requesting a handle to the subscription, the call fails with reason code RC2434 .

If a matching subscription exists and is currently in use by another application, the call fails with RC2429 . If it is currently in use by the same connection the call does not fail and a handle to the subscription is returned.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call fails with RC2523 .

The following tables show the subscription attributes that can be altered by SOALT.

Data type descriptor or function call	Field name	Can this attribute be altered using SOALT?	Reason Code
MQSD	Durability options	No	RC2509
MQSD	Destination Options	Yes	None
MQSD	Registration options	Yes (see note 1)	RC2515 if you try to alter SOGRP
MQSD	Publication options	Yes (see note 2)	None
MQSD	Wildcard options	No	RC2510
MQSD	Other options	No (see note 3)	None
MQSD	ObjectName	No	RC2510
MQSD	SDAU	No (see note 4)	None
MQSD	SDASI	No (see note 4)	None
MQSD	SDEXP	Yes	None
MQSD	SDOS	No	RC2510
MQSD	SDSN	No (see note 5)	None

Data type descriptor or function call	Field name	Can this attribute be altered using SOALT?	Reason Code
MQSD	SDSUD	Yes	None
MQSD	SDCID	Yes (see note 6)	RC2515 when in a grouped subscription
MQSD	SDPRI	Yes	None
MQSD	SDACC	Yes	None
MQSD	SDAID	Yes	None
MQSD	SDSL	No	RC2512
MQSUB	Hobj	Yes (see note 6)	RC2515 when in a grouped subscription

Notes:

1. SOGRP cannot be altered.
2. SONEWP cannot be altered because it is not part of the subscription
3. These options are not part of the subscription
4. This attribute is not part of the subscription
5. This attribute is the identity of the subscription being altered
6. Alterable except when part of a grouped sub (SOGRP)

Durability options: The following options control how durable the subscription is. You can specify only one of these options. If you are altering an existing subscription using the SOALT option, you cannot change the durability of the subscription. On return from an MQSUB call using SORES, the appropriate durability option is set.

SODUR

Request that the subscription to this topic remains until it is explicitly removed using MQCLOSE with the CORMSB option. If this subscription is not explicitly removed it will remain even after this application connects to the queue manager is closed.

If a durable subscription is requested to a topic that is defined as not allowing durable subscriptions, the call fails with RC2436 .

SONDUR

Request that the subscription to this topic is removed when the application connection to the queue manager is closed, if it has not already been explicitly removed. SONDUR is the opposite of the SODUR option, and is defined to aid program documentation. It is the default if neither is specified.

Destination options: The following options control the destination that publications for a topic that has been subscribed to are sent to. If altering an existing subscription using the SOALT option, the destination used for publications for the subscription can be changed. On return from an MQSUB call using SORES, this option is set if appropriate.

SOMAN

Request that the destination that the publications are sent to is managed by the queue manager.

The object handle returned in *HOBj* represents a queue manager managed queue, and is for use with subsequent MQGET, MQCB, MQINQ, or MQCLOSE calls.

An object handle returned from a previous MQSUB call cannot be provided in the *Hobj* parameter when SOMAN is not specified.

Registration options: The following options control the details of the registration that is made to the queue manager for this subscription. If altering an existing subscription using the SOALT option, these

registration options can be changed. On return from an MQSUB call using SORES the appropriate registration options is set.

SOGRP

This subscription is grouped with other subscriptions of the same *SDSL* using the same queue and specifying the same correlation ID so that any publications to topics that would cause more than one publication message to be provided to the group of subscriptions, due to an overlapping set of topic strings being used, only causes one message to be delivered to the queue. If this option is not used, then each unique subscription (identified by *SDSN*) that matches is provided with a copy of the publication which might mean that more than one copy of the publication might be placed on the queue shared by a number of subscriptions.

Only the most significant subscription in the group is provided with a copy of the publication. The most significant subscription is based on the Full topic name up to the point where a wildcard is found. If a mixture of wildcard schemes is used within the group, only the position of the wildcard is important. You are advised not to combine different wildcard schemes within a group of subscriptions that share the same queue.

When creating a new grouped subscription it must still have a unique *SDSN*, but if it matches the full topic name of an existing subscription in the group, the call fails with RC2514 .

If the most significant subscription in group also specifies SONOLC and this is a publication from the same application, then no publication is delivered to the queue.

When altering a subscription made with this option, the fields which imply the grouping, *Hobj* on the MQSUB call (representing the queue and queue manager name), and the *SDCID* cannot be changed. Attempting to alter them causes the call to fail with RC2515 .

This option must be combined with SOSCID with a *SDCID* that is not set to CINONE, and cannot be combined with SOMAN.

SOAUID

When SOAUID is specified, the identity of the subscriber is not restricted to a single user ID. This allows any user to alter or resume the subscription when they have suitable authority. Only a single user can have the subscription at any one time. An attempt to resume use of a subscription currently in use by another application causes the call to fail with RC2429 .

To add this option to an existing subscription, the MQSUB call, using SOALT, must come from the same user ID as the original subscription itself.

If an MQSUB call references an existing subscription with SOAUID set, and the user ID differs from the original subscription, the call succeeds only if the new user ID has authority to subscribe to the topic. On successful completion, future publications to this subscriber are put to the subscriber's queue with the new user ID set in the publication message.

Do not specify both SOAUID and SOFUID. If neither is specified, the default is SOFUID.

SOFUID

When SOFUID is specified, the subscription can be altered or resumed by only the last user ID to alter the subscription. If the subscription has not been altered, it is the user ID that created the subscription.

If an MQSUB verb references an existing subscription with SOAUID set and alters the subscription using SOALT to use the SOFUID, the user ID of the subscription is now fixed at this new user ID. The call succeeds only if the new user ID has authority to subscribe to the topic.

If a user ID other than the one recorded as owning a subscription tries to resume or alter an SOFUID subscription, the call fails with RC2434 . The owning user ID of a subscription can be viewed using the **DISPLAY SBSTATUS** command.

Do not specify both SOAUID and SOFUID. If neither is specified, the default is SOFUID.

Publication options: The following options control the way publications are sent to this subscriber. If altering an existing subscription using the SOALT option, these publication options can be changed.

SONOLC

Tells the broker that the application does not want to see any of its own publications. Publications are considered to have originated from the same application if the connection handles are the same. On return from an MQSUB call using SORES this option is set if appropriate.

SONEWP

No currently retained publications are to be sent, when this subscription is created, only new publications. This option only applies when SOCRE is specified. Any subsequent changes to a subscription do not alter the flow of publications and so any publications that have been retained on a topic, has already been sent to the subscriber as new publications.

If this option is specified without SOCRE it causes the call to fail with RC2046 . On return from an MQSUB call using SORES this option is not set even if the subscription was created using this option.

If this option is not used, previously retained messages are sent to the destination queue provided. If this action fails due to an error, either RC2525 or RC2526 , the creation of the subscription fails.

This option is not valid in combination with SOPUBR.

SOPUBR

Setting this option indicates that the subscriber requests information specifically when required. The queue manager does not send unsolicited messages to the subscriber. The retained publication (or possibly multiple publications if a wildcard is specified in the topic) is sent to the subscriber each time an MQSUBRQ call is made using the Hsub handle from a previous MQSUB call. No publications are sent as a result of the MQSUB call using this option. On return from an MQSUB call using SORES this option is set if appropriate.

This option is not valid in combination with SONEWP.

Wildcard options: The following options control how wildcards are interpreted in the string provided in the *SDOS* field of the MQSD. You can specify only one of these options. If altering an existing subscription using the SOALT option, these wildcard options cannot be changed. On return from an MQSUB call using SORES the appropriate wildcard option is set.

SOWCHR

Wildcards only operate on characters within the topic string. The SOWCHR field treats forward slash (/) as just another character with no special significance.

The behavior defined by SOWCHR is shown in the following table:

Special Character	Behavior
*	Wildcard, zero or more characters
?	Wildcard, one character
%	Escape character to allow the characters '*', '?', or '%' to be used in a string and not be interpreted as a special character, for example, '%*', '%?' or '%%'.

For example, publishing on the following topic:

```
/level0/level1/level2/level3/level4
```

matches subscribers using the following topics:

```
*  
/*  
/ level0/level1/level2/level3/*  
/ level0/level1/*/*level3/level4  
/ level0/level1/le?e12/level3/level4
```

Note: This use of wildcards supplies exactly the meaning provided in IBM MQ V6 and WebSphere MB V6 when using MQRFH1 formatted messages for Publish/Subscribe. It is recommended

that this is not used for newly written applications and is only used for applications that were previously running against that version and have not been changed to use the default wildcard behavior as described in SOWTOP.

SOWTOP

Wildcards only operate on topic elements within the topic string. This is the default behavior if none is chosen.

The behavior required by SOWTOP is shown in the following table:

Special Character	Behavior
/	Topic level separator
#	Wildcard: multiple topic level
+	Wildcard: single topic level

Note:

The '+' and '#' are not treated as wildcards if they are mixed in with other characters (including themselves) within a topic level. In the following string, the '#' and '+' characters are treated as ordinary characters.

```
level0/level1/#+/level3/level#
```

For example, publishing on the following topic:

```
/level0/level1/level2/level3/level4
```

matches subscribers using the following topics:

```
#  
/#  
/ level0/level1/level2/level3/#  
/ level0/level1/+level3/level4
```

Note: This use of wildcards supplies the meaning provided in WebSphere Message Broker Version 6 when using MQRFH2 formatted messages for Publish/Subscribe.

Other options: The following options control the way the API call is issued rather than the subscription. On return from an MQSUB call using SORES these options are unchanged.

SOALTU

The SDAU field contains a user identifier to use to validate this MQSUB call. The call can succeed only if this SDAU is authorized to open the object with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so.

SOSCID

The subscription is to use the correlation identifier supplied in the *SDCID* field. If this option is not specified, a correlation identifier is automatically created by the queue manager at subscription time and is returned to the application in the *SDCID* field. See [SDCID \(24-byte bit string\)SDCID](#) for more information.

SOSETI

The subscription is to use the accounting token and application identity data supplied in the *SDACC* and *SDAID* fields.

If this option is specified, the same authorization check is carried out as if the destination queue was accessed using an MQOPEN call with 00SETI, except in the case where the SOMAN option is also used in which case there is no authorization check on the destination queue.

If this option is not specified, the publications sent to this subscriber has default context information associated with them as follows:

Field in MQMD	Value used
<i>MDUID</i>	The user ID associated with the subscription at the time the subscription was made.
<i>MDACC</i>	Determined from the environment if possible; Set to ACNONE if not.
<i>MDAID</i>	Set to blanks

This option is only valid with SOCRE and SOALT. If used with SORES, the *SDACC* and *SDAID* fields are ignored, so this option has no effect.

If a subscription is altered without using this option where previously the subscription had supplied identity context information, default context information is generated for the altered subscription.

If a subscription allowing different user IDs to use it with option SOAUID, is resumed by a different user ID, default identity context is generated for the new user ID now owning the subscription and any subsequent publications are delivered containing the new identity context.

SOFIQ

The MQSUB call fails if the queue manager is in quiescing state. On z/OS, for a CICS or IMS application, this option also forces the MQSUB call to fail if the connection is in quiescing state.

SDAU (12 byte character string)

If you specify SOALTU, this field contains an alternate user identifier that is used to check the authorization for the subscription and for output to the destination queue (specified in the *Hobj* parameter of the MQSUB call), in place of the user identifier that the application is currently running under.

If successful, the user identifier specified in this field is recorded as the subscription owning user identifier in place of the user identifier that the application is currently running under.

If SOALTU is specified and this field is entirely blank up to the first null character or the end of the field, the subscription can succeed only if no user authorization is must subscribe to this topic with the options specified or the destination queue for output.

If SOALTU is not specified, this field is ignored.

On return from an MQSUB call using SORES, this field is unchanged.

This is an input field. The length of this field is given by LNUID. The initial value of this field is 12 blank characters.

SDPRI (10 digit signed integer)

This is the value that is in the *MQPRI* field of the Message Descriptor (MQMD) of all publication messages matching this subscription. For more information about the *MQPRI* field in the MQMD, see [MDPRI](#).

The value must be greater than or equal to zero; zero is the lowest priority. The following special values can also be used:

PRQDEF

When a subscription queue is provided in the *Hobj* field in the MQSUB call, and is not a managed handle, then the priority for the message is taken from the *DefPriority* attribute of this queue. If the queue so identified is a cluster queue or there is more than one definition in the queue-name resolution path then the priority is determined when the publication message is put to the queue as described for [MDPRI](#).

If the MQSUB call uses a managed handle, the priority for the message is taken from the *DefPriority* attribute of the model queue associated with the topic subscribed to.

PRPUB

The priority for the message is the priority of the original publication. This is the initial value of the field.

If altering an existing subscription using the SOALT option, the *MQPRI* of any future publication messages can be changed.

On return from an MQSUB call using SORES, this field is set to the current priority being used for the subscription.

SDRO (MQCHARV)

SDRO is the long object name after the queue manager resolves the name provided in *SDON*.

If the long object name is provided in *SDOS* and nothing is provided in *SDON*, the value returned in this field is the same as provided in *SDOS*.

If this field is omitted (that is *SDRO.VSBufSize* is zero), the *SDRO* is not returned, but the length is returned in *SDRO.VSLength*. If the length is shorter than the full *SDRO*, it is truncated and returns as many of the rightmost characters as can fit in the provided length.

If *SDRO* is specified incorrectly, according to the description of how to use the [MQCHARV](#) structure, or if it exceeds the maximum length, the call fails with reason code RC2520 .

SDSID (4 byte character string)

This is the structure identifier; the value must be:

SDSIDV

Identifier for Subscription Descriptor structure.

This is always an input field. The initial value of this field is SDSIDV

SDSL (10 digit signed integer)

This is the level associated with the subscription. Publications are only delivered to this subscription if it is in the set of subscriptions with the highest *SDSL* value less than or equal to the *PubLevel* used at publication time.

The value must be in the range zero to 9. Zero is the lowest level.

The initial value of this field is 1.

If altering an existing subscription using the SOALT option, then *SDSL* cannot be changed.

SDSN (MQCHARV)

SDSN specifies the subscription name.

This field is required only if *SDOPT* specifies the SODUR option, but if it is provided it is used by the queue manager for SONDUR as well. If specified, *SDSN* must be unique within the queue manager, because it is the field used to identify subscriptions.

The maximum length of *SDSN* is 10240.

This field serves two purposes. For a SODUR subscription, it is the means by which you identify a subscription to resume it after it has been created, if you have either closed the handle to the subscription (using the COKPSB option) or have been disconnected from the queue manager. Identifying a subscription to remove it after it has been created is done using the MQSUB call with the SORES option. The *SDSN* field is also displayed in the administration view of subscriptions in the *SDSN* field in DISPLAY SBSTATUS.

If *SDSN* is specified incorrectly, according to the description of how to use the [MQCHARV](#) structure, or if it exceeds the maximum length, or if it is omitted when it is required (that is *SDSN.VCHRL* is zero), or if it exceeds the maximum length, the call fails with reason code RC2440 .

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

If altering an existing subscription using the SOALT option, the subscription name cannot be changed, because it is the field used to identify the subscription. It is not changed on output from an MQSUB call with the SORES option.

SDSS (MQCHARV)

SDSS is the string that provides the selection criteria used when subscribing for messages from a topic.

This variable length field is returned on output from an MQSUB call using the SORES option, if a buffer is provided, and if there is also a positive buffer length in VSBufSize. If no buffer is provided on the call, only the length of the selection string is returned in the VSLength field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only VSBufSize bytes are returned in the provided buffer.

If *SDSS* is specified incorrectly, according to the description of how to use the [MQCHARV](#) structure, or if it exceeds the maximum length, the call fails with reason code RC2519.

SDSUD (MQCHARV)

The data provided on the subscription in this field is included as the mq.SubUserData message property of every publication sent to this subscription.

The maximum length of *SDSUD* is 10240.

If *SDSUD* is specified incorrectly, according to the description of how to use the [MQCHARV](#) structure, or if it exceeds the maximum length, the call fails with reason code RC2431.

This is an input field. The initial values of the fields in this structure are the same as those in the MQCHARV structure.

If altering an existing subscription using the SOALT option, the subscription user data can be changed.

This variable length field is returned on output from an MQSUB call using the SORES option, if a buffer is provided and there is a positive buffer length in *VSBufLen*. If no buffer is provided on the call, only the length of the subscription user data is returned in the *VCHRL* field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only *VSBufLen* bytes are returned in the provided buffer.

SDVER (10 digit signed integer)

This is the structure version number; the value must be:

SDVER1

Version-1 Subscription Descriptor structure.

The following constant specifies the version number of the current version:

SDVERC

Current version of Subscription Descriptor structure.

This is always an input field. The initial value of the field is SDVER1.

Initial values

Field name	Name of constant	Value of constant
<i>SDSID</i>	SDSIDV	'SD- - -'
<i>SDVER</i>	SDVER1	1
<i>SDOPT</i>	SONDUR	0
<i>SDON</i>	None	Blanks

Field name	Name of constant	Value of constant
<i>SDAU</i>	None	Blanks
<i>SDASI</i>	SINONE	Nulls
<i>SDEXP</i>	EIULIM	-1
<i>SDOS</i>	Names and values as defined for MQCHARV	
<i>SDSN</i>	Names and values as defined for MQCHARV	
<i>SDSUD</i>	Names and values as defined for MQCHARV	
<i>SDCID</i>	CINONE	Nulls
<i>SDPRI</i>	PRQDEF	-3
<i>SDACC</i>	ACNONE	Nulls
<i>SDAID</i>	None	Blanks
<i>SDSL</i>	None	1
<i>SDRO</i>	Names and values as defined in MQCHARV	
Note: 1. The symbol ~ represents a single blank character.		

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D* MQSD Structure
D*
D* Structure identifier
D  SDSID          1          4
D* Structure version number
D  SDVER          5          8I 0
D* Options associated with subscribing
D  SDOPT          9          12I 0
D* Object name
D  SDON           13          60
D* Alternate user identifier
D  SDAU           61          72
D* Alternate security identifier
D  SDASI          73          112
D* Expiry of Subscription
D  SDEXP         113          116I 0
D* Object Long name
D  SDOSP          117          132*
D  SDOSO          133          136I 0
D  SDOSS          137          140I 0
D  SDOSL          141          144I 0
D  SDOSC          145          148I 0
D* Subscription name
D  SDSNP          149          164*
D  SDSNO          165          168I 0
D  SDSNS          169          172I 0
D  SDSNL          173          176I 0
D  SDSNC          177          180I 0
D* Subscription User data
D  SDSUDP         181          196*
D  SDSUDO         197          200I 0
D  SDSUDS         201          204I 0
D  SDSUDL         205          208I 0
D  SDSUDC         209          212I 0
D* Correlation Id related to this subscription
D  SDCID          213          236

```

D* Priority set in publications		
D SDPRI	237	240I 0
D* Accounting Token set in publications		
D SDACC	241	272
D* Appl Identity Data set in publications		
D SDAID	273	304
D* Message Selector		
D SDSSP	305	320*
D SDSSO	321	324I 0
D SDSSS	325	328I 0
D SDSSL	329	332I 0
D SDSSC	333	336
D* Subscription level		
D SDSL	337	340 0
D* Resolved Long object name		
D SDR0P	341	356*
D SDR00	357	360I 0
D SDR0S	361	364I 0
D SDR0L	365	368I 0
D SDR0C	369	372I 0

MQSMPO - Set message property options

The **MQSMPO** structure allows applications to specify options that control how properties of messages are set.

Overview

Purpose: The structure is an input parameter on the **MQSETMP** call.

Character set and encoding: Data in **MQSMPO** must be in the character set of the application and encoding of the application (ENNAT).

- [“Fields” on page 1183](#)
- [“Initial values” on page 1184](#)
- [“RPG declaration” on page 1185](#)

Fields

The MQSMPO structure contains the following fields; the fields are described in **alphabetical order**:

SPOPT (10-digit signed integer)

Location options: The following options relate to the relative location of the property compared to the property cursor:

SPSETF

Sets the value of the first property that matches the specified name, or if it does not exist, adds a new property after all other properties with a matching hierarchy.

SPSETC

Sets the value of the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired using either the IPINQF or the IPINQN option.

The property cursor is reset when the message handle is reused, or when the message handle is specified in the *HMSG* field of the MQGMO structure on an MQGET call or the MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established or if the property pointed to by the property cursor has been deleted, the call fails with completion code CCFAIL and reason code RC2471.

SPSETA

Sets a new property after the property pointed to by the property cursor. The property pointed to by the property cursor is the one that was last inquired using either the IPINQF or the IPINQO option.

The property cursor is reset when the message handle is reused, or when the message handle is specified in the *HMSG* field of the MQGMO structure on an MQGET call or the MQPMO structure on an MQPUT call.

If this option is used when the property cursor has not yet been established or if the property pointed to by the property cursor has been deleted, the call fails with completion code CCFAIL and reason code RC2471.

If you need none of the options described, use the following option:

SPNONE

No options specified.

This is always an input field. The initial value of this field is SPSETF.

SPSID (10-digit signed integer)

This is the structure identifier; the value must be:

SPSIDV

Identifier for set message property options structure.

This is always an input field. The initial value of this field is **SPSIDV**.

SPVAKCSI (10-digit signed integer)

The character set of the property value to be set if the value is a character string.

This is always an input field. The initial value of this field is **CSAPL**.

SPVALENC (10-digit signed integer)

The encoding of the property value to be set if the value is numeric.

This is always an input field. The initial value of this field is **ENNAT**.

SPVER (10-digit signed integer)

This is the structure version number; the value must be:

SPVER1

Version-1 set message property options structure.

The following constant specifies the version number of the current version:

SPVERC

Current version of set message property options structure.

This is always an input field. The initial value of this field is **SPVER1**.

Initial values

<i>Table 613. Initial values of fields in MQSMPO</i>		
Field name	Name of constant	Value of constant
<i>SPSID</i>	SPSIDV	'SMPO'
<i>SPVER</i>	SPVER1	1
<i>SPOPT</i>	SPNONE	0
<i>SPVALENC</i>	ENNAT	Depends on environment
<i>SPVALCSI</i>	CSAPL	-3

RPG declaration

```
D* MQSMPO Structure
D*
D*
D* Structure identifier
D  SPSID              1          4    INZ('SMPO')
D*
D* Structure version number
D  SPVER              5          8I 0 INZ(1)
D*
D* Options that control the action of
D* MQSETMP
D  SPOPT              9         12I 0 INZ(0)
D*
D* Encoding of Value
D  SPVALENC          13         16I 0 INZ(273)
D*
D* Character set identifier of Value
D  SPVALCSI          17         20I 0 INZ(-3)
```

MQSRO - Subscription Request Options

The MQSRO structure allows the application to specify options that control how a subscription request is made.

Overview

Purpose: The structure is an input/output parameter on the MQSUBRQ call.

Version: The current version of MQSRO is SRVER1.

- [“Fields” on page 1185](#)
- [“Initial values” on page 1186](#)
- [“RPG declaration” on page 1186](#)

Fields

The MQSRO structure contains the following fields; the fields are described in **alphabetical order**:

SRNMP (10-digit signed integer)

This is an output field, returned to the application to indicate the number of publications sent to the subscription queue as a result of this call. Although this number of publications have been sent as a result of this call, there is no guarantee that this many messages will be available for the application to get, especially if they are non-persistent messages.

There may be more than one publication if the topic subscribed to, contained a wildcard. If no wildcards were present in the topic string when the subscription represented by *HSUB* was created, then at most one publication is sent as a result of this call.

SROPT (10-digit signed integer)

One of the following options must be specified. Only one option can be specified.

Other options: The following option controls what happens when the queue manager is quiescing:

SRFIQ

The MQSUBRQ call fails if the queue manager is in the quiescing state.

Default option: If the option described previously is not required, the following option must be used:

SRNONE

Use this value to indicate that no other options have been specified; all options assume their default values.

SRNONE helps program documentation. Although it is not intended that this option be used with any other, because its value is zero, this use cannot be detected.

SRSID (4-byte character string)

This is the structure identifier; the value must be:

SRSIDV

Identifier for Subscription Request SROPT structure.

This is always an input field. The initial value of this field is SRSIDV.

SERVER (10-digit signed integer)

This is the structure version number; the value must be:

SERVER1

Version-1 Subscription Request Options structure.

The following constant specifies the version number of the current version:

SERVERC

Current version of Subscription Request Options structure.

This is always an input field. The initial value of this field is SERVER1.

Initial values

Field name	Name of constant	Value of constant
<i>SRSID</i>	SRSIDV	'SR0¬'
<i>SERVER</i>	SERVER1	1
<i>SROPT</i>	SRNONE	0
<i>SRNMP</i>	None	0
Notes: 1. The symbol ¬ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages.		

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..  
D* MQSRO Structure  
D*  
D* Structure identifier  
D  SRSID          1          4  
D* Structure version number  
D  SERVER         5          8I 0  
D* Options that control the action of MQSUBRQ  
D  SROPT          9         12I 0  
D* Number of publications sent  
D  SRNMP         13         16I 0
```

MQSTS - Status reporting structure

The MQSTS structure describes the data in the status structure returned by the MQSTAT command.

Overview

Character set and encoding: Character data in MQSTS is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in MQSTS is in the native machine encoding; this is given by *ENNAT*.

Usage: The MQSTAT command is used to retrieve status information. This information is returned in an MQSTS structure. For information about MQSTAT, see [“MQSTAT - Retrieve status information”](#) on page 1314.

- [“Fields”](#) on page 1187
- [“Initial values”](#) on page 1190
- [“RPG declaration”](#) on page 1190

Fields

The MQSTS structure contains the following fields; the fields are described in **alphabetical order**:

STSCC (10-digit signed integer)

This is the completion code resulting from the first error reported in the MQSTS structure.

This is always an output field. The initial value of this field is CCOK.

STSFC (10-digit signed integer)

This is the number of asynchronous put calls that failed.

This is an output field. The initial value of this field is 0.

STSOBJN (48-byte character string)

This is the local name of the object involved in the first failure.

This is an output field. The initial value of this field is 48 blank characters.

STSOQMGR (48-byte character string)

This is the name of the queue manager on which the *STSOBJN* object is defined. A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected (the local queue manager).

This is an output field. The initial value of this field is 48 blank characters.

STS00 (10-digit signed integer)

The STS00 used to open the object being reported upon. Present only in Version 2 of MQSTS or higher.

The value of STS00 depends on the value of the MQSTAT STYPE parameter.

STATAPT

Zero.

STATREC

Zero.

STATRER

The STS00 used when the failure occurred. The reason for the failure is reported in the *STSCC* and *STSRC* fields in the MQSTS structure.

STS00 is an output field. Its initial value is zero.

STSOS (MQCHARV)

Long object name of failing object being reported on. Present only in Version 2 of MQSTS or higher.

STSOS is a MQCHARV field with a maximum length of 10240. See [MQCHARV](#) for a description of how to use the MQCHARV structure.

The interpretation of STSOS depends on the value of the MQSTAT STYPE parameter.

STATAPT

This is the long object name of the queue or topic used in the MQPUT operation, which failed.

STATREC

Zero length string

STATRER

This is the long object name of the object that caused the reconnection to fail.

STSOS is an output field. Its initial value is a zero length string.

STSOT (10-digit signed integer)

The type of object being named in *ObjectName*. Possible values are:

OTALSQ

Alias queue.

OTLOCQ

Local queue.

OTMODQ

Model queue.

OTQ

Queue.

OTREMQ

Remote queue.

OTTOP

Topic.

This is always an output field. The initial value of this field is OTQ.

STSRC (10-digit signed integer)

This is the reason code resulting from the first error reported in the MQSTS structure

This is always an output field. The initial value of this field is RCNONE.

STSR OBJN (48-byte character string)

This is the name of the destination queue named in *STSOBJN* after the local queue manager resolves the name. The name returned is the name of a queue that exists on the queue manager identified by *STSRQMGR*.

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *STSR OBJN* is set to blanks:

- A topic
- A queue, but not opened for browse, input, or output

This is an output field. The initial value of this field is 48 blank characters.

STSRQMGR (48-byte character string)

This is the name of the destination queue manager after the local queue manager resolves the name. The name returned is the name of the queue manager that owns the queue identified by *STSR OBJN*. *STSRQMGR* can be the name of the local queue manager.

If *STSR OBJN* is a shared queue that is owned by the queue-sharing group to which the local queue manager belongs, *STSRQMGR* is the name of the queue-sharing group. If the queue is owned by some other queue-sharing group, *STSR OBJN* can be the name of the queue-sharing group or the name of a queue manager that is a member of the queue-sharing group (the nature of the value returned is determined by the queue definitions that exist at the local queue manager).

A nonblank value is returned only if the object is a single queue opened for browse, input, or output (or any combination). If the object opened is any of the following, *STSRQMGR* is set to blanks:

- A topic
- A queue, but not opened for browse, input, or output

- A cluster queue with OOBNDN specified (or with OOBNDQ in effect when the *DefBind* queue attribute has the value OOBNDN)

This is an output field. The initial value of this field is 48 blank characters.

STSSC (10-digit signed integer)

This is the number of asynchronous put calls that succeeded.

This is an output field. The initial value of this field is 0.

STSSID (4-byte character string)

This is the structure identifier. The value must be:

STSSID

Identifier for status reporting structure.

The initial value of this field is STSSID.

STSSO (10 digit signed integer)

The STSSO used to open the failing subscription. Present only in Version 2 of MQSTS or higher.

The interpretation of STSSO depends on the value of the MQSTAT STYPE parameter.

STATAPT

Zero.

STATREC

Zero.

STATRER

The STSSO used when the failure occurred. The reason for the failure is reported in the *STSCC* and *STSRC* fields in the MQSTS structure. If the failure is not related to subscribing to a topic, the value returned is zero.

STSSO is an output field. Its initial value is zero.

STSSUN (MQCHARV)

The name of the failing subscription. Present only in Version 2 of MQSTS or higher.

STSSUN is a MQCHARV field with a maximum length of 10240. See [MQCHARV](#) for a description of how to use the MQCHARV structure.

The interpretation of STSSUN depends on the value of the MQSTAT STYPE parameter.

STATAPT

Zero length string.

STATREC

Zero length string.

STATRER

The name of the subscription that caused reconnection to fail. If no subscription name is available, or the failure is not related to a subscription, this is a zero-length string.

STSSUN is an output field. Its initial value is a zero length string.

STSVER (10-digit signed integer)

This is the structure version number. The value must be:

STSVER1

Version number for status reporting structure.

The following constant specifies the version number of the current version:

STSVRC

Current version of status reporting structure.

The initial value of this field is STSVR1.

STSWC (10-digit signed integer)

This is the number of asynchronous put calls that completed with a warning.

This is an output field. The initial value of this field is 0.

Initial values

Table 614. Initial values of fields in MQSTS		
Field name	Name of constant	Value of constant
STSSID	STSID	
STSVR	STSVRC	STSVR1
STSCC	CCOK	0
STSRC	RCNONE	0
STSSC	None	0
STSWC	None	0
STSFC	None	0
STSOT	None	0
STSOBJN	None	Blanks
STSOQMGR	None	Blanks
STSR0BJN	None	Blanks
STSRQMGR	None	Blanks
STSOS	Names and values as defined for MQCHARV	
STSSUN	Names and values as defined for MQCHARV	
STS00	None	0
STSS0	None	0

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..
D* MQSTS Structure
D*
D* Structure identifier
D STSSID 1 4
D* Structure version number
D STSVR 5 8I 0
D* Completion code
D STSCC 9 12I 0
D* Reason code
D STSRC 13 16I 0
D* Success count
D STSSC 17 20I 0
D* Warning count
D STSWC 21 24I 0
D* Failure count
D STSFC 25 28I 0
D* Object type
D STSOT 29 32I 0
D* Object name
D STSOBJN 33 80
```

```

D* Object queue manager
D STSQMGR      81      128
D* Resolved object name
D STSROBJN    129      176
D* Resolved object queue manager name
D STSRQMGR    177      224
D* Ver:1 **
D* Failing object long name
D* Address of variable length string
D STSOSCHRP   225      240*
D* Offset of variable length string
D STSOSCHRO   241      244I 0
D* Size of buffer
D STSOSVSBS   245      248I 0
D* Length of variable length string
D STSOSCHRL   249      252I 0
D* CCSID of variable length string
D STSOSCHRC   253      256I 0
D* Failing subscription name
D* Address of variable length string
D STSSUNCHRP  257      272*
D* Offset of variable length string
D STSSUNCHRO  273      276I 0
D* Size of buffer
D STSSUNVSBS  277      280I 0
D* Length of variable length string
D STSSUNCHRL  281      284I 0
D* CCSID of variable length string
D STSSUNCHRC  285      288I 0
D* Failing open options
D STS00       289      292I 0
D* Failing subscription options
D STSS0       293      296I 0
D* Ver:2 **

```

MQTM - Trigger message

The MQTM structure describes the data in the trigger message that is sent by the queue manager to a trigger-monitor application when a trigger event occurs for a queue.

Overview

Purpose: This structure is part of the IBM MQ Trigger Monitor Interface (TMI), which is one of the IBM MQ framework interfaces.

Format name: FMTM.

Character set and encoding: Character data in MQTM is in the character set of the queue manager that generates the MQTM. Numeric data in MQTM is in the machine encoding of the queue manager that generates the MQTM.

The character set and encoding of the MQTM are given by the *MDCSI* and *MDENC* fields in:

- The MQMD (if the MQTM structure is at the start of the message data), or
- The header structure that precedes the MQTM structure (all other cases).

Usage: A trigger-monitor application may need to pass some or all of the information in the trigger message to the application which is started by the trigger-monitor application. Information which may be needed by the started application includes *TMQN*, *TMTD*, and *TMUD*. The trigger-monitor application can pass the MQTM structure directly to the started application, or pass an MQTMC2 structure instead, depending on what is permitted by the environment and convenient for the started application. For information about MQTMC2, see [“MQTMC2 - Trigger message 2 \(character format\)” on page 1195](#).

- On IBM i, the trigger-monitor application provided with IBM MQ passes an MQTMC2 structure to the started application.

For information about triggers, see [Prerequisites for triggering](#).

- [“MQMD for a trigger message” on page 1192](#)
- [“Fields” on page 1193](#)

- [“Initial values” on page 1195](#)
- [“RPG declaration” on page 1195](#)

MQMD for a trigger message

MQMD for a trigger message: The fields in the MQMD of a trigger message generated by the queue manager are set as follows:

Field in MQMD	Value used
<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER1
<i>MDREP</i>	RONONE
<i>MDMT</i>	MTDGRM
<i>MDEXP</i>	EIULIM
<i>MDFB</i>	FBNONE
<i>MDENC</i>	ENNAT
<i>MDCSI</i>	Queue manager's <i>CodedCharSetId</i> attribute
<i>MDFMT</i>	FMTM
<i>MDPRI</i>	Initiation queue's <i>DefPriority</i> attribute
<i>MDPER</i>	PENPER
<i>MDMID</i>	A unique value
<i>MDCID</i>	CINONE
<i>MDBOC</i>	0
<i>MDRQ</i>	Blanks
<i>MDRM</i>	Name of queue manager
<i>MDUID</i>	Blanks
<i>MDACC</i>	ACNONE
<i>MDAID</i>	Blanks
<i>MDPAT</i>	ATQM, or as appropriate for the message channel agent
<i>MDPAN</i>	First 28 bytes of the queue manager name
<i>MDPD</i>	Date when trigger message is sent
<i>MDPT</i>	Time when trigger message is sent
<i>MDAOD</i>	Blanks

An application that generates a trigger message is recommended to set similar values, except for the following:

- The *MDPRI* field can be set to PRQDEF (the queue manager will change this to the default priority for the initiation queue when the message is put).
- The *MDRM* field can be set to blanks (the queue manager will change this to the name of the local queue manager when the message is put).
- The context fields should be set as appropriate for the application.

Fields

The MQTM structure contains the following fields; the fields are described in **alphabetical order**:

TMAI (256-byte character string)

Application identifier.

This is a character string that identifies the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *AppLId* attribute of the process object identified by the *TMPN* field; see [“Attributes for process definitions” on page 1353](#) for details of this attribute. The content of this data is of no significance to the queue manager.

The meaning of *TMAI* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ requires *TMAI* to be the name of an executable program.

The length of this field is given by LNPROA. The initial value of this field is 256 blank characters.

TMAT (10-digit signed integer)

Application type.

This identifies the nature of the program to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *AppLType* attribute of the process object identified by the *TMPN* field; see [“Attributes for process definitions” on page 1353](#) for details of this attribute. The content of this data is of no significance to the queue manager.

TMAT can have one of the following standard values. User-defined types can also be used, but should be restricted to values in the range ATUFST through ATULST:

ATCICS

CICS transaction.

ATVSE

CICS/VSE transaction.

AT400

IBM i application.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

The initial value of this field is 0.

TMED (128-byte character string)

Environment data.

This is a character string that contains environment-related information pertaining to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *EnvData* attribute of the process object identified by the *TMPN* field; see [“Attributes for process definitions” on page 1353](#) for details of this attribute. The content of this data is of no significance to the queue manager.

The length of this field is given by LNPROE. The initial value of this field is 128 blank characters.

TMPN (48-byte character string)

Name of process object.

This is the name of the queue manager process object specified for the triggered queue, and can be used by the trigger-monitor application that receives the trigger message. The queue manager

initializes this field with the value of the *ProcessName* attribute of the queue identified by the *TMQN* field; see [“Attributes for queues” on page 1324](#) for details of this attribute.

Names that are shorter than the defined length of the field are always padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by LNPRON. The initial value of this field is 48 blank characters.

TMQN (48-byte character string)

Name of triggered queue.

This is the name of the queue for which a trigger event occurred, and is used by the application started by the trigger-monitor application. The queue manager initializes this field with the value of the *QName* attribute of the triggered queue; see [“Attributes for queues” on page 1324](#) for details of this attribute.

Names that are shorter than the defined length of the field are padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

TMSID (4-byte character string)

Structure identifier.

The value must be:

TMSIDV

Identifier for trigger message structure.

The initial value of this field is TMSIDV.

TMTD (64-byte character string)

Trigger data.

This is free-format data for use by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *TriggerData* attribute of the queue identified by the *TMQN* field; see [“Attributes for queues” on page 1324](#) for details of this attribute. The content of this data is of no significance to the queue manager.

The length of this field is given by LNTRGD. The initial value of this field is 64 blank characters.

TMUD (128-byte character string)

User data.

This is a character string that contains user information relevant to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *UserData* attribute of the process object identified by the *TPPN* field; see [“Attributes for process definitions” on page 1353](#) for details of this attribute. The content of this data is of no significance to the queue manager.

The length of this field is given by LNPROU. The initial value of this field is 128 blank characters.

TMVER (10-digit signed integer)

Structure version number.

The value must be:

TMVER1

Version number for trigger message structure.

The following constant specifies the version number of the current version:

TMVERC

Current version of trigger message structure.

The initial value of this field is TMVER1.

Initial values

Table 615. Initial values of fields in MQTM		
Field name	Name of constant	Value of constant
TMSID	TMSIDV	'TM '
TMVER	TMVER1	1
TMQN	None	Blanks
TMPN	None	Blanks
TMTD	None	Blanks
TMAT	None	0
TMAI	None	Blanks
TMED	None	Blanks
TMUD	None	Blanks
Notes:		
1. The symbol represents a single blank character.		

RPG declaration

```
D*.1....:....2....:....3....:....4....:....5....:....6....:....7..  
D*  
D* MQTM Structure  
D*  
D* Structure identifier  
D TMSID 1 4 INZ('TM  ' )  
D* Structure version number  
D TMVER 5 8I 0 INZ(1)  
D* Name of triggered queue  
D TMQN 9 56 INZ  
D* Name of process object  
D TMPN 57 104 INZ  
D* Trigger data  
D TMTD 105 168 INZ  
D* Application type  
D TMAT 169 172I 0 INZ(0)  
D* Application identifier  
D TMAI 173 428 INZ  
D* Environment data  
D TMED 429 556 INZ  
D* User data  
D TMUD 557 684 INZ
```

MQTMC2 - Trigger message 2 (character format)

When a trigger-monitor application retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor might need to pass some or all of the information in the trigger message to the application that is started by the trigger monitor.

Overview

Purpose: Information that may be needed by the started application includes *TC2QN*, *TC2TD*, and *TC2UD*. The trigger monitor application can pass the MQTM structure directly to the started application, or pass an MQTMC2 structure instead, depending on what is permitted by the environment and convenient for the started application.

This structure is part of the IBM MQ Trigger Monitor Interface (TMI), which is one of the IBM MQ framework interfaces.

Character set and encoding: Character data in MQTMC2 is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue manager attribute.

Usage: The MQTMC2 structure is like the format of the MQTM structure. The difference is that the non-character fields in MQTM are changed in MQTMC2 to character fields of the same length, and the queue manager name is added at the end of the structure.

- On IBM i, the trigger monitor application provided with IBM MQ passes an MQTMC2 structure to the started application.
- [“Fields” on page 1196](#)
- [“Initial values” on page 1197](#)
- [“RPG declaration” on page 1197](#)

Fields

The MQTMC2 structure contains the following fields; the fields are described in **alphabetical order**:

TC2AI (256-byte character string)

Application identifier.

See the *TMAI* field in the MQTM structure.

TC2AT (4-byte character string)

Application type.

This field always contains blanks, whatever the value in the *TMAT* field in the MQTM structure of the original trigger message.

TC2ED (128-byte character string)

Environment data.

See the *TMED* field in the MQTM structure.

TC2PN (48-byte character string)

Name of process object.

See the *TPPN* field in the MQTM structure.

TC2QMN (48-byte character string)

Queue manager name.

This is the name of the queue manager at which the trigger event occurred.

TC2QN (48-byte character string)

Name of triggered queue.

See the *TMQN* field in the MQTM structure.

TC2SID (4-byte character string)

Structure identifier.

The value must be:

TCSIDV

Identifier for trigger message (character format) structure.

TC2TD (64-byte character string)

Trigger data.

See the *TMTD* field in the MQTM structure.

TC2UD (128-byte character string)

User data.

See the *TMUD* field in the MQTM structure.

TC2VER (4-byte character string)

Structure version number.

The value must be:

TCVER2

Version 2 trigger message (character format) structure.

The following constant specifies the version number of the current version:

TCVERC

Current version of trigger message (character format) structure.

Initial values

Table 616. Initial values of fields in MQTMC2		
Field name	Name of constant	Value of constant
TC2SID	TCSIDV	'TMC¬'
TC2VER	TCVER2	'¬¬¬2'
TC2QN	None	Blanks
TC2PN	None	Blanks
TC2TD	None	Blanks
TC2AT	None	Blanks
TC2AI	None	Blanks
TC2ED	None	Blanks
TC2UD	None	Blanks
TC2QMN	None	Blanks
Notes:		
1. The symbol ¬ represents a single blank character.		

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQTMC2 Structure
D*
D* Structure identifier
D TC2SID                1         4
D* Structure version number
D TC2VER                5         8
D* Name of triggered queue
D TC2QN                 9        56
D* Name of process object
D TC2PN                57       104
D* Trigger data
D TC2TD               105       168
D* Application type
D TC2AT               169       172
D* Application identifier
D TC2AI               173       428
D* Environment data
D TC2ED               429       556
D* User data
```

D	TC2UD	557	684
D*	Queue manager name		
D	TC2QMN	685	732

MQWIH - Work information header

The MQWIH structure describes the information that must be present at the start of a message that is to be handled by the z/OS workload manager.

Overview

Format name: FMWIH.

Character set and encoding: The fields in the MQWIH structure are in the character set and encoding given by the *MDCSI* and *MDENC* fields in the header structure that precedes MQWIH, or by those fields in the MQMD structure if the MQWIH is at the start of the application message data.

The character set must be one that has single-byte characters for the characters that are valid in queue names.

Usage: If a message is to be processed by the z/OS workload manager, the message must begin with an MQWIH structure.

- [“Fields” on page 1198](#)
- [“Initial values” on page 1200](#)
- [“RPG declaration” on page 1200](#)

Fields

The MQWIH structure contains the following fields; the fields are described in **alphabetical order**:

WICSI (10-digit signed integer)

Character-set identifier of data that follows MQWIH.

This specifies the character set identifier of the data that follows the MQWIH structure; it does not apply to character data in the MQWIH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The following special value can be used:

CSINHT

Inherit character-set identifier of this structure.

Character data in the data *following* this structure is in the same character set as this structure.

The queue manager changes this value in the structure sent in the message to the actual character-set identifier of the structure. Provided no error occurs, the value CSINHT is not returned by the MQGET call.

CSINHT cannot be used if the value of the *MDPAT* field in MQMD is ATBRKR.

The initial value of this field is CSUNDF.

WIENC (10-digit signed integer)

Numeric encoding of data that follows MQWIH.

This specifies the numeric encoding of the data that follows the MQWIH structure; it does not apply to numeric data in the MQWIH structure itself.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data.

The initial value of this field is 0.

WIFLG (10-digit signed integer)

Flags

The value must be:

WINONE

No flags.

The initial value of this field is WINONE.

WIFMT (8-byte character string)

Format name of data that follows MQWIH.

This specifies the format name of the data that follows the MQWIH structure.

On the MQPUT or MQPUT1 call, the application must set this field to the value appropriate to the data. The rules for coding this field are the same as those for the *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

WILEN (10-digit signed integer)

Length of MQWIH structure.

The value must be:

WILEN1

Length of version-1 work information header structure.

The following constant specifies the length of the current version:

WILENC

Length of current version of work information header structure.

The initial value of this field is WILEN1.

WIRSV (32-byte character string)

Reserved.

This is a reserved field; it must be blank.

WISID (4-byte character string)

Structure identifier.

The value must be:

WISIDV

Identifier for work information header structure.

The initial value of this field is WISIDV.

WISNM (32-byte character string)

Service name.

This is the name of the service that is to process the message.

The length of this field is given by LNSVNM. The initial value of this field is 32 blank characters.

WISST (8-byte character string)

Service step name.

This is the name of the step of *WISNM* to which the message relates.

The length of this field is given by LNSVST. The initial value of this field is 8 blank characters.

WITOK (16-byte bit string)

Message token.

This is a message token that uniquely identifies the message.

For the MQPUT and MQPUT1 calls, this field is ignored. The length of this field is given by LNMTOK. The initial value of this field is MTKNON.

WIVER (10-digit signed integer)

Structure version number.

The value must be:

WIVER1

Version-1 work information header structure.

The following constant specifies the version number of the current version:

WIVERC

Current version of work information header structure.

The initial value of this field is WIVER1.

Initial values

Table 617. Initial values of fields in MQWIH		
Field name	Name of constant	Value of constant
WISID	WISIDV	'WIH~'
WIVER	WIVER1	1
WILEN	WILEN1	120
WIENC	None	0
WICSI	CSUNDF	0
WIFMT	FMNONE	Blanks
WIFLG	WINONE	0
WISNM	None	Blanks
WISST	None	Blanks
WITOK	MTKNON	Nulls
WIRSV	None	Blanks
Notes:		
1. The symbol ~ represents a single blank character.		

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D*
D* MQWIH Structure
D*
D* Structure identifier
D WISID 1 4 INZ('WIH ')
D* Structure version number
D WIVER 5 8I 0 INZ(1)
D* Length of MQWIH structure
D WILEN 9 12I 0 INZ(120)
D* Numeric encoding of data that followsMQWIH
D WIENC 13 16I 0 INZ(0)
D* Character-set identifier of data thatfollows MQWIH
D WICSI 17 20I 0 INZ(0)
D* Format name of data that followsMQWIH
D WIFMT 21 28 INZ(' ')
D* Flags
D WIFLG 29 32I 0 INZ(0)
D* Service name
D WISNM 33 64 INZ
D* Service step name
D WISST 65 72 INZ
```

D* Message token			
D WITOK	73	88	INZ(X'0000000000000000-0000000000000000')
D			
D* Reserved			
D WIRSV	89	120	INZ

MQXQH - Transmission-queue header

The MQXQH structure describes the information that is prefixed to the application message data of messages when they are on transmission queues.

Overview

Purpose: A transmission queue is a special type of local queue that temporarily holds messages destined for remote queues (that is, destined for queues that do not belong to the local queue manager). A transmission queue is denoted by the *Usage* queue attribute having the value USTRAN.

Format name: FMXQH.

Character set and encoding: Data in MQXQH must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT for the C programming language.

The character set and encoding of the MQXQH must be set into the *MDCSI* and *MDENC* fields in:

- The separate MQMD (if the MQXQH structure is at the start of the message data), or
- The header structure that precedes the MQXQH structure (all other cases).

Usage: A message that is on a transmission queue has *two* message descriptors:

- One message descriptor is stored separately from the message data; this is called the *separate message descriptor*, and is generated by the queue manager when the message is placed on the transmission queue. Some of the fields in the separate message descriptor are copied from the message descriptor provided by the application on the MQPUT or MQPUT1 call.

The separate message descriptor is the one that is returned to the application in the *MSGDSC* parameter of the MQGET call when the message is removed from the transmission queue.

- A second message descriptor is stored within the MQXQH structure as part of the message data; this is called the *embedded message descriptor*, and is a copy of the message descriptor that was provided by the application on the MQPUT or MQPUT1 call (with minor variations).

The embedded message descriptor is always a version-1 MQMD. If the message put by the application has nondefault values for one or more of the version-2 fields in the MQMD, an MQMDE structure follows the MQXQH, and is in turn followed by the application message data (if any). The MQMDE is either:

- Generated by the queue manager (if the application uses a version-2 MQMD to put the message), or
- Already present at the start of the application message data (if the application uses a version-1 MQMD to put the message).

The embedded message descriptor is the one that is returned to the application in the *MSGDSC* parameter of the MQGET call when the message is removed from the final destination queue.

- [“Fields in the separate message descriptor” on page 1202](#)
- [“Fields in the embedded message descriptor” on page 1203](#)
- [“Putting messages on remote queues” on page 1203](#)
- [“Putting messages directly on transmission queues” on page 1203](#)
- [“Getting messages from transmission queues” on page 1204](#)
- [“Fields” on page 1204](#)
- [“Initial values” on page 1205](#)
- [“RPG declaration” on page 1205](#)

Fields in the separate message descriptor

The fields in the separate message descriptor are set by the queue manager as shown in the following list. If the queue manager does not support the version-2 MQMD, a version-1 MQMD is used without loss of function.

Field in separate MQMD	Value used
<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER2
<i>MDREP</i>	Copied from the embedded message descriptor, but with the bits identified by ROAUXM set to zero. (This prevents a COA or COD report message being generated when a message is placed on or removed from a transmission queue.)
<i>MDMT</i>	Copied from the embedded message descriptor.
<i>MDEXP</i>	Copied from the embedded message descriptor.
<i>MDFB</i>	Copied from the embedded message descriptor.
<i>MDENC</i>	ENNAT
<i>MDCSI</i>	Queue manager's <i>CodedCharSetId</i> attribute.
<i>MDFMT</i>	FMXQH
<i>MDPRI</i>	Copied from the embedded message descriptor.
<i>MDPER</i>	Copied from the embedded message descriptor.
<i>MDMID</i>	A new value is generated by the queue manager. This message identifier is different from the <i>MDMID</i> that the queue manager may have generated for the embedded message descriptor (see described previously).
<i>MDCID</i>	The <i>MDMID</i> from the embedded message descriptor.
<i>MDBOC</i>	0
<i>MDRQ</i>	Copied from the embedded message descriptor.
<i>MDRM</i>	Copied from the embedded message descriptor.
<i>MDUID</i>	Copied from the embedded message descriptor.
<i>MDACC</i>	Copied from the embedded message descriptor.
<i>MDAID</i>	Copied from the embedded message descriptor.
<i>MDPAT</i>	ATQM
<i>MDPAN</i>	First 28 bytes of the queue manager name.
<i>MDPD</i>	Date when message was put on transmission queue.
<i>MDPT</i>	Time when message was put on transmission queue.
<i>MDAOD</i>	Blanks
<i>MDGID</i>	GINONE
<i>MDSEQ</i>	1
<i>MDOFF</i>	0
<i>MDMFL</i>	MFNONE
<i>MDOLN</i>	OLUNDF

Fields in the embedded message descriptor

The fields in the embedded message descriptor have the same values as those in the *MSGDSC* parameter of the *MQPUT* or *MQPUT1* call, except for the following:

- The *MDVER* field always has the value *MDVER1*.
- If the *MDPRI* field has the value *PRQDEF*, it is replaced by the value of the queue's *DefPriority* attribute.
- If the *MDPER* field has the value *PEQDEF*, it is replaced by the value of the queue's *DefPersistence* attribute.
- If the *MDMID* field has the value *MINONE*, or the *PMNMID* option was specified, or the message is a distribution-list message, *MDMID* is replaced by a new message identifier generated by the queue manager.

When a distribution-list message is split into smaller distribution-list messages placed on different transmission queues, the *MDMID* field in each of the new embedded message descriptors is the same as that in the original distribution-list message.

- If the *PMNCID* option was specified, *MDCID* is replaced by a new correlation identifier generated by the queue manager.
- The context fields are set as indicated by the *PM** options specified in the *PMO* parameter; the context fields are:
 - *MDACC*
 - *MDAID*
 - *MDAOD*
 - *MDPAN*
 - *MDPAT*
 - *MDPD*
 - *MDPT*
 - *MDUID*
- The version-2 fields (if they were present) are removed from the *MQMD*, and moved into an *MQMDE* structure, if one or more of the version-2 fields has a nondefault value.

Putting messages on remote queues

: When an application puts a message on a remote queue (either by specifying the name of the remote queue directly, or by using a local definition of the remote queue), the local queue manager:

- Creates an *MQXQH* structure containing the embedded message descriptor
- Appends an *MQMDE* if one is needed and is not already present
- Appends the application message data
- Places the message on an appropriate transmission queue

Putting messages directly on transmission queues

It is also possible for an application to put a message directly on a transmission queue. In this case the application must prefix the application message data with an *MQXQH* structure, and initialize the fields with appropriate values. In addition, the *MDFMT* field in the *MSGDSC* parameter of the *MQPUT* or *MQPUT1* call must have the value *FMXQH*.

Character data in the *MQXQH* structure created by the application must be in the character set of the local queue manager (defined by the *CodedCharSetId* queue manager attribute), and integer data must be in the native machine encoding. In addition, character data in the *MQXQH* structure must be padded with blanks to the defined length of the field; the data must not be ended prematurely by using a null

character, because the queue manager does not convert the null and subsequent characters to blanks in the MQXQH structure.

Note however that the queue manager does not check that an MQXQH structure is present, or that valid values have been specified for the fields.

Getting messages from transmission queues

Applications that get messages from a transmission queue must process the information in the MQXQH structure in an appropriate fashion. The presence of the MQXQH structure at the beginning of the application message data is indicated by the value FMXQH being returned in the *MDFMT* field in the *MSGDSC* parameter of the MQGET call. The values returned in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter, indicates the character set and encoding of the character and integer data in the MQXQH structure. The character set and encoding of the application message data are defined by the *MDCSI* and *MDENC* fields in the embedded message descriptor.

Fields

The MQXQH structure contains the following fields; the fields are described in **alphabetical order**:

XQMD (MQMD1)

Original message descriptor.

This is the embedded message descriptor, and is a close copy of the message descriptor MQMD that was specified as the *MSGDSC* parameter on the MQPUT or MQPUT1 call when the message was originally put to the remote queue.

Note: This is a version-1 MQMD.

The initial values of the fields in this structure are the same as those in the MQMD structure.

XQRQ (48-byte character string)

Name of destination queue.

This is the name of the message queue that is the apparent eventual destination for the message (this may prove not to be the actual eventual destination if, for example, this queue is defined at *XQRQM* to be a local definition of another remote queue).

If the message is a distribution-list message (that is, the *MDFMT* field in the embedded message descriptor is FMDH), *XQRQ* is blank.

The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

XQRQM (48-byte character string)

Name of destination queue manager.

This is the name of the queue manager or queue-sharing group that owns the queue that is the apparent eventual destination for the message.

If the message is a distribution-list message, *XQRQM* is blank.

The length of this field is given by LNQM. The initial value of this field is 48 blank characters.

XQSID (4-byte character string)

Structure identifier.

The value must be:

XQSIDV

Identifier for transmission-queue header structure.

The initial value of this field is XQSIDV.

XQVER (10-digit signed integer)

Structure version number.

The value must be:

XQVER1

Version number for transmission-queue header structure.

The following constant specifies the version number of the current version:

XQVERC

Current version of transmission-queue header structure.

The initial value of this field is XQVER1.

Initial values

Table 618. Initial values of fields in MQXQH		
Field name	Name of constant	Value of constant
XQSID	XQSIDV	'XQH¬'
XQVER	XQVER1	1
XQRQ	None	Blanks
XQRQM	None	Blanks
XQMD	Same names and values as MQMD; see Table 598 on page 1109	-
Notes:		
1. The symbol ¬ represents a single blank character.		

RPG declaration

```
D*.1.....2.....3.....4.....5.....6.....7..
D*
D* MQXQH Structure
D*
D* Structure identifier
D  XQSID          1          4    INZ('XQH ')
D* Structure version number
D  XQVER          5          8I 0  INZ(1)
D* Name of destination queue
D  XQRQ           9          56    INZ
D* Name of destination queue manager
D  XQRQM          57         104    INZ
D* Original message descriptor
D  XQ1SID         105         108    INZ('MD ')
D  XQ1VER         109         112I 0  INZ(1)
D  XQ1REP         113         116I 0  INZ(0)
D  XQ1MT          117         120I 0  INZ(8)
D  XQ1EXP         121         124I 0  INZ(-1)
D  XQ1FB          125         128I 0  INZ(0)
D  XQ1ENC         129         132I 0  INZ(273)
D  XQ1CSI         133         136I 0  INZ(0)
D  XQ1FMT         137         144    INZ(' ')
D  XQ1PRI         145         148I 0  INZ(-1)
D  XQ1PER         149         152I 0  INZ(2)
D  XQ1MID         153         176    INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D  XQ1CID         177         200    INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D  XQ1BOC         201         204I 0  INZ(0)
D  XQ1RQ          205         252    INZ
D  XQ1RM          253         300    INZ
D  XQ1UID         301         312    INZ
```

D	XQ1ACC	313	344	INZ(X'0000000000000000-
D				0000000000000000000000-
D				0000000000000000000000-
D				000000')
D	XQ1AID	345	376	INZ
D	XQ1PAT	377	380I 0	INZ(0)
D	XQ1PAN	381	408	INZ
D	XQ1PD	409	416	INZ
D	XQ1PT	417	424	INZ
D	XQ1AOD	425	428	INZ

Function calls

Use this information to learn about the function calls available in IBM i programming.

Conventions used in the call descriptions on IBM i

For each call, this collection of topics gives a description of the parameters and usage of the call. This is followed by typical invocations of the call, and typical declarations of its parameters, in the RPG programming language.

Important: When coding IBM MQ API calls you must ensure that all relevant parameters (as described in the following sections) are provided. Failure to do so can produce unpredictable results.

The description of each call contains the following sections:

Call name

The call name, followed by a brief description of the purpose of the call.

Parameters

For each parameter, the name is followed by its data type in parentheses () and its direction; for example:

CMPCOD (9-digit decimal integer) - output

There is more information about the structure data types in [“Elementary data types” on page 959](#).

The direction of the parameter can be:

Input

You (the programmer) must provide this parameter.

Output

The call returns this parameter.

Input/output

You must provide this parameter, but it is modified by the call.

There is also a brief description of the purpose of the parameter, together with a list of any values that the parameter can take.

The last two parameters in each call are a completion code and a reason code. The completion code indicates whether the call completed successfully, partially, or not at all. Further information about the partial success or the failure of the call is given in the reason code.

Usage notes

Additional information about the call, describing how to use it and any restrictions on its use.

RPG invocation

Typical invocation of the call, and declaration of its parameters, in RPG.

Other notational conventions are:

Constants

Names of constants are shown in uppercase; for example, OOOUT.


Arrays

In some calls, parameters are arrays of character strings with a size that is not fixed. In the descriptions of these parameters, a lowercase *n* represents a numeric constant. When you code the declaration for that parameter, replace the *n* with the numeric value you require.

MQBACK - Back out changes

The MQBACK call indicates to the queue manager that all of the message gets and puts that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work are deleted; messages retrieved as part of a unit of work are reinstated on the queue.

- On IBM i, this call is not supported for applications running in compatibility mode.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

- [“Syntax” on page 1207](#)
- [“Usage notes” on page 1207](#)
- [“Parameters” on page 1208](#)
- [“RPG Declaration” on page 1209](#)

Syntax

MQBACK (*Hconn*, *CompCode*, *Reason*)

Usage notes

Consider these usage notes when using MQBACK.

1. This call can be used only when the queue manager itself coordinates the unit of work. This is a local unit of work, where the changes affect only IBM MQ resources.
2. In environments where the queue manager does not coordinate the unit of work, the appropriate back-out call must be used instead of MQBACK. The environment may also support an implicit back out caused by the application terminating abnormally.
 - On IBM i, this call can be used for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
3. If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager” on page 1246](#) for further details.
4. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

The queue manager keeps *three* sets of group and segment information, one set for each of the following:

- The last successful MQPUT call (this can be part of a unit of work).
- The last successful MQGET call that removed a message from the queue (this can be part of a unit of work).

- The last successful MQGET call that browsed a message on the queue (this *cannot* be part of a unit of work).

If the application puts or gets the messages as part of a unit of work, and the application then decides to back out the unit of work, the group and segment information is restored to the value that it had previously:

- The information associated with the MQPUT call is restored to the value that it had before the first successful MQPUT call for that queue handle in the current unit of work.
- The information associated with the MQGET call is restored to the value that it had before the first successful MQGET call for that queue handle in the current unit of work.

Queues which were updated by the application after the unit of work had started, but outside the scope of the unit of work, do not have their group and segment information restored if the unit of work is backed out.

Restoring the group and segment information to its previous value when a unit of work is backed out allows the application to spread a large message group or large logical message consisting of many segments across several units of work, and to restart at the correct point in the message group or logical message if one of the units of work fails. Using several units of work might be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point if a system failure occurs. For details of how to restart at the correct point after a system failure, see the PMLOGO option described in [“MQPMO - Put-message options” on page 1131](#), and the GMLOGO option described in [“MQGMO - Get-message options” on page 1035](#).

The remaining usage notes apply only when the queue manager coordinates the units of work:

1. A unit of work has the same scope as a connection handle. This means that all IBM MQ calls which affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *HCONN* parameter described in [“MQCONN - Connect queue manager” on page 1232](#) for information about the scope of connection handles.
2. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
3. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but which never issues a commit or backout call, can cause queues to fill up with messages that are not available to other applications. To guard against this possibility, the administrator should set the *MaxUncommittedMsgs* queue manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.

Parameters

The MQBACK call has the following parameters:

HCONN (10-digit signed integer) - input

Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *COMCOD*.

If *COMCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *COMCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2101

(2101, X'835') Object damaged.

RC2123

(2123, X'84B') Result of commit or back-out operation is mixed.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..  
C                                CALLP      MQBACK(HCONN : COMCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..  
DMQBACK          PR          EXTPROC('MQBACK')  
D* Connection handle  
D HCONN          10I 0 VALUE  
D* Completion code  
D COMCOD          10I 0  
D* Reason code qualifying COMCOD  
D REASON          10I 0
```

IBM i

AIX, HP-UX, IBM i, Solaris, Windows S/390

MQBEGIN - Begin unit of work

The MQBEGIN call begins a unit of work that is coordinated by the queue manager, and that may involve external resource managers.

- This call is supported in the following environments: AIX, HP-UX, IBM i, Solaris, Windows.
- [“Syntax” on page 1210](#)
- [“Usage notes” on page 1210](#)
- [“Parameters” on page 1211](#)

- [“RPG Declaration” on page 1212](#)

Syntax

MQBEGIN (*HCONN*, *BEGOP*, *CMPCOD*, *REASON*)

Usage notes

1. The MQBEGIN call can be used to start a unit of work that is coordinated by the queue manager and that might involve changes to resources owned by other resource managers. The queue manager supports three types of unit-of-work:

Queue-manager-coordinated local unit of work

This is a unit of work in which the queue manager is the only resource manager participating, and so the queue manager acts as the unit-of-work coordinator.

- To start this type of unit of work, the PMSYP or GMSYP option should be specified on the first MQPUT, MQPUT1, or MQGET call in the unit of work.

It is not necessary for the application to issue the MQBEGIN call to start the unit of work, but if MQBEGIN is used, the call completes with CCWARN and reason code RC2121.

- To commit or back out this type of unit of work, the MQCMIT or MQBACK call must be used.

Queue-manager-coordinated global unit of work

This is a unit of work in which the queue manager acts as the unit-of-work coordinator, both for IBM MQ resources *and* for resources belonging to other resource managers. Those resource managers cooperate with the queue manager to ensure that all changes to resources in the unit of work are committed or backed out together.

- To start this type of unit of work, the MQBEGIN call must be used.
- To commit or back out this type of unit of work, the MQCMIT and MQBACK calls must be used.

Externally-coordinated global unit of work

This is a unit of work in which the queue manager is a participant, but the queue manager does not act as the unit-of-work coordinator. Instead, there is an external unit-of-work coordinator with whom the queue manager cooperates.

- To start this type of unit of work, the relevant call provided by the external unit-of-work coordinator must be used.

If the MQBEGIN call is used to try to start the unit of work, the call fails with reason code RC2012.

- To commit or back out this type of unit of work, the commit and back-out calls provided by the external unit-of-work coordinator must be used.

If the MQCMIT or MQBACK call is used to try to commit or back out the unit of work, the call fails with reason code RC2012.

2. If the application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager” on page 1246](#) for further details.
3. An application can participate in only one unit of work at a time. The MQBEGIN call fails with reason code RC2128 if there is already a unit of work in existence for the application, regardless of which type of unit of work it is.
4. The MQBEGIN call is not valid in an IBM MQ client environment. An attempt to use the call fails with reason code RC2012.
5. When the queue manager is acting as the unit-of-work coordinator for global units of work, the resource managers that can participate in the unit of work are defined in the queue manager's configuration file.
6. On IBM i, the three types of unit of work are supported as follows:

- **Queue-manager-coordinated local units of work** can be used only when a commitment definition does not exist at the job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
- **Queue-manager-coordinated global units of work** are not supported.
- **Externally-coordinated global units of work** can be used only when a commitment definition exists at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must have been issued for the job. If this has been done, the IBM i COMMIT and ROLLBACK operations apply to IBM MQ resources as well as to resources belonging to other participating resource managers.

Parameters

The MQBEGIN call has the following parameters:

HCONN (10-digit signed integer) - input

Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

BEGOP (MQBO) - input/output

Options that control the action of MQBEGIN.

See [“MQBO - Begin options” on page 980](#) for details.

If no options are required, programs written in C or S/390 assembler can specify a null parameter address, instead of specifying the address of an MQBO structure.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2121

(2121, X'849') No participating resource managers registered.

RC2122

(2122, X'84A') Participating resource manager not available.

If *CMPCOD* is CCFAIL:

RC2134

(2134, X'856') Begin-options structure not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2012

(2012, X'7DC') Call not valid in environment.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RC2128

(2128, X'850') Unit of work already started.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQBEGIN(HCONN : BEGOP : CMPCOD :
C                                REASON)

```

The prototype definition for the call is:

```

D*.1.....2.....3.....4.....5.....6.....7..
D*MQBEGIN      PR      EXTPROC('MQBEGIN')
D* Connection handle
D HCONN              10I 0 VALUE
D* Options that control the action of MQBEGIN
D BEGOP              12A
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CMPCOD
D REASON              10I 0

```

MQBUFMH - Convert buffer into message handle

The MQBUFMH function call converts a buffer into a message handle and is the inverse of the MQMHBUF call.

This call takes a message descriptor and MQRFH2 properties in the buffer and makes them available through a message handle. The MQRFH2 properties in the message data are, optionally, removed. The *Encoding*, *CodedCharSetId*, and *Format* fields of the message descriptor are updated, if necessary, to correctly describe the contents of the buffer after the properties have been removed.

- [“Syntax” on page 1212](#)
- [“Usage notes” on page 1213](#)
- [“Parameters” on page 1213](#)
- [“RPG Declaration” on page 1214](#)

Syntax

MQBUFMH (*Hconn*, *Hmsg*, *BufMsgHOpts*, *MsgDesc*, *Buffer*, *BufferLength*, *DataLength*, *CompCode*, *Reason*)

Usage notes

MQBUFMH calls cannot be intercepted by API exits - a buffer is converted into a message handle in the application space; the call does not reach the queue manager.

Parameters

The MQBUFMH call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *HCONN* must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created by using HCUNAS, a valid connection must be established on the thread converting a buffer into a message handle. If a valid connection is not established, the call fails with RC2009.

HMSG (20-digit signed integer) - input

This handle is the message handle for which a buffer is required. The value was returned by a previous MQCRTMH call.

BMHOPT (MQBMHO) - input

The MQBMHO structure allows applications to specify options that control how message handles are produced from buffers.

See [“MQBMHO - Buffer to message handle options” on page 979](#) for details.

MSGDSC (MQMD) - input/output

The *MSGDSC* structure contains the message descriptor properties and describes the contents of the buffer area.

On output from the call, the properties are optionally removed from the buffer area and, in this case, the message descriptor is updated to correctly describe the buffer area.

Data in this structure must be in the character set and encoding of the application.

BUFLEN (10-digit signed integer) - input

BUFLEN is the length of the Buffer area, in bytes.

A *BUFLEN* of zero bytes is valid, and indicates that the buffer area contains no data.

BUFFER (1-byte bit string x BUFLEN) - input/output

BUFFER defines the area containing the message buffer. For most data, you must align the buffer on a 4-byte boundary.

If *BUFFER* contains character or numeric data, set the *CodedCharSetId* and *Encoding* fields in the *MSGDSC* parameter to the values appropriate to the data; this enables the data to be converted, if necessary.

If properties are found in the message buffer they are optionally removed; they later become available from the message handle on return from the call.

In the C programming language, the parameter is declared as a pointer-to-void, which means the address of any type of data can be specified as the parameter.

If the *BUFLEN* parameter is zero, *BUFFER* is not referred to. In this case, the parameter address passed by programs written in C or System/390 assembler can be null.

DATLEN (10-digit signed integer) - output

DATLEN is the length, in bytes, of the buffer which might have the properties removed.

CMPCOD (10-digit signed integer) - output

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2489

(2489, X'09B9') Buffer to message handle options structure not valid.

RC2004

(2004, X'07D4') Buffer parameter not valid.

RC2005

(2005, X'07D5') Buffer length parameter not valid.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2009

(2009, X'07D9') Connection to queue manager lost.

RC2460

(2460, X'099C') Message handle not valid.

RC2026

(2026, X'07EA') Message descriptor not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2334

(2334, X'091E') MQRFH2 structure not valid.

RC2421

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..  
C                                CALLP      MQBUFMH(HCONN : HMSG : BMHOPT :  
                                MSGDSC : BUFLN : BUFFER :  
                                DATLEN : CMPCOD : REASON)
```

The prototype definition for the call is:

```

DMQBUFMH          PR          EXTPROC('MQBUFMH')
D* Connection handle
D HCONN              10I 0
D* Message handle
D HMSG              10I 0
D* Options that control the action of MQBUFMH
D BMHOPT            12A  VALUE
D* Message descriptor
D MSGDSC              364A
D* Length in bytes of the Buffer area
D BUFLN              10I 0
D* Area to contain the message buffer
D BUFFER              *  VALUE
D* Length of the output buffer
D DATLEN              10I 0
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CompCode
D REASON              10I 0

```

MQCB - Manage callback

The MQCB call reregisters a callback for the specified object handle and controls activation and changes to the callback.

A callback is a piece of code (specified as either the name of a function that can be dynamically linked or as a function pointer) that is called by IBM MQ when certain events occur.

To use MQCB and MQCTL on a V7 client you must be connected to a V7 server and the **SHARECNV** parameter of the channel must have a non-zero value.

For information about Global units of work see: [Global units of work](#).

The types of callback that can be defined are:

Message consumer

A message consumer callback function is called when a message, meeting the selection criteria specified, is available on an object handle.

Only one callback function can be registered against each object handle. If a single queue is to be read with multiple selection criteria then the queue must be opened multiple times and a consumer function registered on each handle.

Event handler

The event handler is called for conditions that affect the whole callback environment.

The function is called when an event condition occurs, for example, a queue manager or connection stopping or quiescing.

The function is not called for conditions that are specific to a single message consumer, for example RC2016; it is called however if a callback function does not end normally.

- [“Syntax” on page 1215](#)
- [“Usage notes for MQCB” on page 1215](#)
- [“Parameters for MQCB” on page 1217](#)
- [“RPG Declaration” on page 1223](#)

Syntax

MQCB (*HCONN*, *OPERATN*, *HOBJ*, *CBDS*, *MSGDSC*, *GMO*, *CMPCOD*, *REASON*)

Usage notes for MQCB

1. MQCB is used to define the action to be invoked for each message, matching the specified criteria, available on the queue. When the action is processed, either the message is removed from the queue

and passed to the defined message consumer, or a message token is provided, which is used to retrieve the message.

2. MQCB can be used to define callback routines before starting consumption with MQCTL or it can be used from within a callback routine.
3. To use MQCB from outside of a callback routine, you must first suspend message consumption by using MQCTL and resume consumption afterward.

Message consumer callback sequence

You can configure a consumer to invoke callback at key points during the lifecycle of the consumer. For example:

- when the consumer is first registered,
- when the connection is started,
- when the connection is stopped and
- when the consumer is deregistered, either explicitly, or implicitly by an MQCLOSE.

Table 619. MQCTL verb definitions	
Verb	Meaning
MQCTL(START)	MQCTL call by using the CTLSR Operation
MQCTL(STOP)	MQCTL call by using the CTLSP Operation
MQCTL(WAIT)	MQCTL call by using the CTLSW Operation

Allows the consumer to maintain state associated with the consumer. When a callback is requested by an application, the rules for consumer invocation are as follows:

REGISTER

Is always the first type of invocation of the callback.

Is always called on the same thread as the MQCB(CBREG) call.

START

Is always called synchronously with the MQCTL(START) verb.

- All START callbacks are completed before the MQCTL(START) verb returns.

Is on the same thread as the message delivery if CTLTHR is requested.

The call with start is not guaranteed if, for example, a previous callback issues MQCTL(STOP) during the MQCTL(START).

STOP

No further messages or events are delivered after this call until the connection is restarted.

A STOP is guaranteed if the application was previously called for START, or a message, or an event.

DEREGISTER

Is always the last type of invocation of the callback.

Ensure that your application performs thread-based initialization and cleanup in the START and STOP callbacks. You can do non thread-based initialization and cleanup with REGISTER and DEREGISTER callbacks.

Do not make any assumptions about the life and availability of the thread other than what is stated. For example, do not rely on a thread staying alive beyond the last call to DEREGISTER. Similarly, when you have chosen not to use CTLTHR, do not assume that the thread exists whenever the connection is started.

If your application has particular requirements for thread characteristics, it can always create a thread accordingly, then use MQCTL(WAIT). This step *donates* the thread to IBM MQ for asynchronous message delivery.

Message consumer connection usage

Normally, when an application issues another MQI call while one is outstanding, the call fails with reason code RC2219.

There are special cases, however, when the application must issue a further MQI call before the previous call has completed. For example, the consumer can be invoked during an MQCB call with CBRE.

In such an instance, when as a result of the application issuing either an MQCB or MQCTL verb, the application is called back, the application is allowed to issue a further MQI call. This instance means you can issue, for example, an MQOPEN call, in the consumer function when called with a CBCCALLT type of CBCTRC. Any MQI call, except for MQDISC, is allowed.

Parameters for MQCB

The MQCB call has the following parameters:

HCONN (10-digit signed integer) - input


Manage callback function - HCONN parameter.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and you can specify the following special value for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

OPERATN (10-digit signed integer) - input

Manage callback function - OPERATN parameter.

The operation being processed on the callback defined for the specified object handle. You must specify one of the following options; if more than one option is required, the values can be added (do not add the same constant more than once) or combined by using the bitwise OR operation (if the programming language supports bit operations).

Combinations that are not valid are noted; all other combinations are valid.

CBREG

Define the callback function for the specified object handle. This operation defines the function to be called and the selection criteria to be used.

If a callback function is already defined for the object handle the definition is replaced. If an error is detected while replacing the callback, the function is deregistered.

If a callback is registered in the same callback function in which it was previously deregistered, this is treated as a replace operation; any initial or final calls are not invoked.

You can use CBREG with CTLSU or CTLRE.

CBUNR

Stop the consuming of messages for the object handle and removes the handle from those eligible for a callback.

A callback is automatically deregistered if the associated handle is closed.

If CBUNR is called from within a consumer, and the callback has a stop call defined, it is invoked upon return from the consumer.

If this operation is issued against an *Hobj* with no registered consumer, the call returns with RC2448.

CTLSU

Suspends the consuming of messages for the object handle.

If this operation is applied to an event handler, the event handler does not get events while suspended, and any events missed while in the suspended state are not provided to the operation when it is resumed.

While suspended, the consumer function continues to get the control type callbacks.

CTLRE

Resume the consuming of messages for the object handle.

If this operation is applied to an event handler, the event handler does not get events while suspended, and any events missed while in the suspended state are not provided to the operation when it is resumed.

CBDSC (MQCBD) - input

Manage callback function - CBDSC parameter.

This is a structure that identifies the callback function that is being registered by the application and the options used when registering it.

See “MQCBD - Callback descriptor” on page 278 for details of the structure.

Callback descriptor is required only for the CBREG option; if the descriptor is not required, the parameter address passed can be null.

HOBJ (10-digit signed integer) - input

Manage callback function - HOBJ parameter.

This handle represents the access that has been established to the object from which a message is to be consumed. This is a handle that has been returned from a previous [MQOPEN](#) or [MQSUB](#) call (in the *HOBJ* parameter).

HOBJ is not required when defining an event handler routine (CBTEH) and must be specified as HONONE.

If this *Hobj* has been returned from an [MQOPEN](#) call, the queue must have been opened with one or more of the following options:

- OOINPS
- OOINPX
- OOINPQ
- OOBWR

MSGDSC (MQMD) - input

Manage callback function -MSGDSC parameter.

This structure describes the attributes of the message required, and the attributes of the message retrieved.

The *MsgDesc* parameter defines the attributes of the messages required by the consumer, and the version of the MQMD to be passed to the message consumer.

The *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* in the MQMD are used for message selection, depending on the options specified in the *GetMsgOpts* parameter.

The *Encoding* and *CodedCharSetId* are used for message conversion if you specify the GMCONV option.

See [MQMD](#) for details.

MsgDesc is used only for CBREG and, if you require values other than the default for any fields. *MsgDesc* is not used for an event handler.

If the descriptor is not required the parameter address passed can be null.

Note, that if multiple consumers are registered against the same queue with overlapping selectors, the chosen consumer for each message is undefined.

GMO (MQGMO) - input

Manage callback function - GMO parameter.

Options that control how the message consumer gets messages.

All options have the meaning as described in [“MQGMO - Get-message options”](#) on page 1035, when used on an MQGET call, except:

GMSSIG

This option is not permitted.

GMBRWF, GMBRWN, GMMBH, GMMBC

The order of messages delivered to a browsing consumer is dictated by the combinations of these options. Significant combinations are:

GMBRWF

The first message on the queue is delivered repeatedly to the consumer. This is useful when the consumer destructively consumes the message in the callback. Use this option with care.

GMBRWN

The consumer is given each message on the queue, from the current cursor position until the end of the queue is reached.

GMBRWF + GMBRWN

The cursor is reset to the start of the queue. The consumer is then given each message until the cursor reaches the end of the queue.

GMBRWF + GMMBH or GMMBC

Starting at the beginning of the queue, the consumer is given the first nonmarked message on the queue, which is then marked for this consumer. This combination ensures that the consumer can receive new messages added behind the current cursor point.

GMBRWN + GMMBH or GMMBC

Starting at the cursor position the consumer is given the next nonmarked message on the queue, which is then marked for this consumer. Use this combination with care because messages can be added to the queue behind the current cursor position.

GMBRWF + GMBRWN + GMMBH or GMMBC

This combination is not permitted, if used the call returns RC2046.

GMNWT, GMWT and GMWI

These options control how the consumer is invoked.

GMNWT

The consumer is never called with RC2033. The consumer is only invoked for messages and events

GMWT with a zero GMWI

The RC2033 code is only passed to the consumer when there are no messages and

- the consumer has been started
- the consumer has been delivered at least one message since the last no messages reason code.

This prevents the consumer from polling in a busy loop when a zero wait interval is specified.

GMWT and a positive GMWI

The user is invoked after the specified wait interval with reason code RC2033. This call is made regardless of whether any messages have been delivered to the consumer. This allows the user to perform heartbeat or batch type processing.

GMWT and GMWI of WIULIM

This specifies an infinite wait before returning RC2033. The consumer is never called with RC2033.

GMO is used only for CBREG and, if you require values other than the default for any fields. *GMO* is not used for an event handler.

If the options are not required the parameter address passed can be null.

If a message properties handle is provided in the MQGMO structure, a copy is provided in the MQGMO structure that is passed into the consumer callback. On return from the MQCB call, the application can delete the message properties handle.

CMPCOD (10-digit signed integer) - output

Manage callback function - CMPCOD parameter.

The completion code; it is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Manage callback function - REASON parameter.

The following reason codes are the codes that the queue manager can return for the *REASON* parameter.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CompCode* is CCFAIL:

RC2204

(2204, X'89C') Adapter not available.

RC2133

(2133, X'855') Unable to load data conversion services modules.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2374

(2374, X'946') API exit failed.

RC2183

(2183, X'887') Unable to load API exit.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2219

(2219, X'8AB') MQI call entered before previous call complete.

RC2487

(2487, X'9B7') Incorrect callback type field.

RC2448

(2448, X'990') Unable to deregister, suspend, or resume because there is no registered callback.

RC2486
(2486, X'9B6') Either *CallbackFunction* or *CallbackName* must be specified but not both.

RC2483
(2483, X'9B3') Incorrect callback type field.

RC2484
(2484, X'9B4') Incorrect MQCBD options field.

RC2140
(2140, X'85C') Wait request rejected by CICS.

RC2009
(2009, X'7D9') Connection to queue manager lost.

RC2217
(2217, X'8A9') Not authorized for connection.

RC2202
(2202, X'89A') Connection quiescing.

RC2203
(2203, X'89B') Connection shutting down.

RC2207
(2207, X'89F') Correlation-identifier error.

RC2010
(2010, X'7DA') Data length parameter not valid.

RC2016
(2016, X'7E0') Gets inhibited for the queue.

RC2351
(2351, X'92F') Global units of work conflict.

RC2186
(2186, X'88A') Get-message options structure not valid.

RC2353
(2353, X'931') Handle in use for global unit of work.

RC2018
(2018, X'7E2') Connection handle not valid.

RC2019
(2019, X'7E3') Object handle not valid.

RC2259
(2259, X'8D3') Inconsistent browse specification.

RC2245
(2245, X'8C5') Inconsistent unit-of-work specification.

RC2246
(2246, X'8C6') Message under cursor not valid for retrieval.

RC2352
(2352, X'930') Global unit of work conflicts with local unit of work.

RC2247
(2247, X'8C7') Match options not valid.

RC2485
(2485, X'9B4') Incorrect *MaxMsgLength* field.

RC2026
(2026, X'7EA') Message descriptor not valid.

RC2497
(2497, X'9C1') The specified function entry point could not be found in the module.

RC2496

(2496, X'9C0') Module found, however it is of the wrong type; not 32 bit, 64 bit, or a valid dynamic link library.

RC2495

(2495, X'9BF') Module not found in the search path or not authorized to load.

RC2250

(2250, X'8CA') Message sequence number not valid.

RC2331

(2331, X'91B') Use of message token not valid.

RC2033

(2033, X'7F1') No message available.

RC2034

(2034, X'7F2') Browse cursor not positioned on message.

RC2036

(2036, X'7F4') Queue not open for browse.

RC2037

(2037, X'7F5') Queue not open for input.

RC2041

(2041, X'7F9') Object definition changed since opened.

RC2101

(2101, X'835') Object damaged.

RC2206

(2206, X'89E') Incorrect operation code on API Call.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2193

(2193, X'891') Error accessing page-set data set.

RC2052

(2052, X'804') Queue has been deleted.

RC2394

(2394, X'95A') Queue has wrong index type.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2161

(2161, X'871') Queue manager quiescing.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2069

(2069, X'815') Signal outstanding for this handle.

RC2071

(2071, X'817') Insufficient storage available.

RC2109

(2109, X'83D') Call suppressed by exit program.

RC2024

(2024, X'7E8') No more messages can be handled within current unit of work.

RC2072

(2072, X'818') Syncpoint support not available.

RC2195

(2195, X'893') Unexpected error occurred.

RC2354

(2354, X'932') Enlistment in global unit of work failed.

RC2355

(2355, X'933') Mixture of unit-of-work calls not supported.

RC2255

(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2090

(2090, X'82A') Wait interval in MQGMO not valid.

RC2256

(2256, X'8D0') Wrong version of MQGMO supplied.

RC2257

(2257, X'8D1') Wrong version of MQMD supplied.

RC2298

(2298, X'8FA') The function requested is not available in the current environment.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQCB(HCONN : OPERATN : CBDSC :
                                           HOBJ : MSGDSC : GMO :
                                           DATLEN : CMPCOD : REASON)

```

The prototype definition for the call is:

DMQCB	PR	EXTPROC('MQCB')
D* Connection handle		
D HCONN		10I 0 VALUE
D* Operation		
D OPERATN		10I 0 VALUE
D* Callback descriptor		
D CBDSC		180A
D* Object handle		
D HOBJ		10I 0 VALUE
D* Message Descriptor		
D MSGDSC		364A
D* Get options		
D GMO		112A
D* Completion code		
D CMPCOD		10I 0
* Reason code qualifying CompCode		
D REASON		10I 0

MQCLOSE - Close object

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN call.

- [“Syntax” on page 1223](#)
- [“Usage notes” on page 1224](#)
- [“Parameters” on page 1225](#)
- [“RPG Declaration” on page 1229](#)

Syntax

MQCLOSE (*HCONN*, *HOBJ*, *OPTS*, *CMPCOD*, *REASON*)

Usage notes

1. When an application issues the MQDISC call, or ends either normally or abnormally, any objects that were opened by the application and are still open are closed automatically with the CONONE option.
2. The following points apply if the object being closed is a *queue*:
 - If operations on the queue are performed as part of a unit of work, the queue can be closed before or after the syncpoint occurs without affecting the outcome of the syncpoint.
 - If the queue was opened with the OOBROW option, the browse cursor is destroyed. If the queue is later reopened with the OOBROW option, a new browse cursor is created (see the OOBROW option described in MQOPEN).
 - If a message is currently locked for this handle at the time of the MQCLOSE call, the lock is released (see the GMLK option described in [“MQGMO - Get-message options” on page 1035](#)).
3. The following points apply if the object being closed is a *dynamic queue* (either permanent or temporary):

- For a dynamic queue, the options CODEL or COPURG can be specified regardless of the options specified on the corresponding MQOPEN call.
- When a dynamic queue is deleted, all MQGET calls with the GMWT option that are outstanding against the queue are canceled and reason code RC2052 is returned. See the GMWT option described in [“MQGMO - Get-message options” on page 1035](#).

After a dynamic queue has been deleted, any call (other than MQCLOSE) that attempts to reference the queue using a previously acquired *HOBJ* handle fails with reason code RC2052.

Be aware that although a deleted queue cannot be accessed by applications, the queue is not removed from the system, and associated resources are not freed, until all handles that reference the queue have been closed, and all units of work that affect the queue have been either committed or backed out.

- When a permanent dynamic queue is deleted, if the *HOBJ* handle specified on the MQCLOSE call is *not* the one that was returned by the MQOPEN call that created the queue, a check is made that the user identifier which was used to validate the MQOPEN call is authorized to delete the queue. If the OOALTU option was specified on the MQOPEN call, the user identifier checked is the *ODAU*.

This check is not performed if:

- The handle specified is the one returned by the MQOPEN call that created the queue.
- The queue being deleted is a temporary dynamic queue.
- When a temporary dynamic queue is closed, if the *HOBJ* handle specified on the MQCLOSE call is the one that was returned by the MQOPEN call that created the queue, the queue is deleted. This occurs regardless of the close options specified on the MQCLOSE call. If there are messages on the queue, they are discarded; no report messages are generated.

If there are uncommitted units of work that affect the queue, the queue and its messages are still deleted, but this does not cause the units of work to fail. However, as described previously, the resources associated with the units of work are not freed until each of the units of work has been either committed or backed out.

4. The following points apply if the object being closed is a *distribution list*:

- The only valid close option for a distribution list is CONONE; the call fails with reason code RC2046 or RC2045 if any other options are specified.
- When a distribution list is closed, individual completion codes and reason codes are not returned for the queues in the list - only the *CMPCOD* and *REASON* parameters of the call are available for diagnostic purposes.


If a failure occurs closing one of the queues, the queue manager continues processing and attempts to close the remaining queues in the distribution list. The *CMPCOD* and *REASON* parameters of the call are then set to return information describing the failure. Thus it is possible for the completion

code to be CCFAIL, even though most of the queues were closed successfully. The queue that encountered the error is not identified.

If there is a failure on more than one queue, it is not defined which failure is reported in the *CMPCOD* and *REASON* parameters.

5. On IBM i, if the application was connected implicitly when the first MQOPEN call was issued, an implicit MQDISC occurs when the last MQCLOSE is issued.

Only applications running in compatibility mode can be connected implicitly; other applications must issue the MQCONN or MQCONNX call to connect to the queue manager explicitly.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Parameters

The MQCLOSE call has the following parameters:

HCONN (10-digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HOBJ (10-digit signed integer) - input/output

Object handle.

This handle represents the object that is being closed. The object can be of any type. The value of *HOBJ* was returned by a previous MQOPEN call.

On successful completion of the call, the queue manager sets this parameter to a value that is not a valid handle for the environment. This value is:

HOUNUH

Unusable object handle.

OPTS (10-digit signed integer) - input

Options that control the action of MQCLOSE.

The *OPTS* parameter controls how the object is closed. Only permanent dynamic queues and subscriptions can be closed in more than one way. Permanent dynamic queues can either be retained or deleted; these are queues with a *DefinitionType* attribute that has the value QDPERM (see the *DefinitionType* attribute described in [“Attributes for queues” on page 1324](#)). The close options are summarized in a table later in this topic.

Durable subscriptions can either be kept or removed; these are created using the MQSUB call with the SODUR option.

When closing the handle to a managed destination (that is the *Hobj* parameter returned on an MQSUB call which used the SOMAN option) the queue manager will clean up any unretrieved publications when the associated subscription has also been removed. That is done using the CORMSB option on the *Hsub* parameter returned on an MQSUB call. Note that CORMSB is the default behavior on MQCLOSE for a non-durable subscription.

When closing a handle to a non-managed destination you are responsible for cleaning up the queue where publications are sent. You are recommended to close the subscription using CORMSB first and then process messages off the queue until there are none remaining.

One (and only one) of the following must be specified:

Dynamic queue closure options

These options control how permanent dynamic queues are closed:

CODEL

Delete the queue.

The queue is deleted if either of the following is true:

- It is a permanent dynamic queue, created by a previous MQOPEN call, and there are no messages on the queue and no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *HOBj*. In this case, all the messages on the queue are purged.

In all other cases, including the case where the *Hobj* was returned on an MQSUB call, the call fails with reason code RC2045, and the object is not deleted.

COPURG

Delete the queue, purging any messages on it.

The queue is deleted if either of the following is true:

- It is a permanent dynamic queue, created by a previous MQOPEN call, and there are no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *HOBj*.

In all other cases, including the case where the *Hobj* was returned on an MQSUB call, the call fails with reason code RC2045, and the object is not deleted.

The next table shows which close options are valid, and whether the object is retained or deleted.

Table 620. Valid close options for use with retained or deleted objects			
Type of object or queue	CONONE	CODEL	COPURG
Object other than a queue	Retained	Not valid	Not valid
Predefined queue	Retained	Not valid	Not valid
Permanent dynamic queue	Retained	Deleted if empty and no pending updates	Messages deleted; queue deleted if no pending updates
Temporary dynamic queue (call issued by creator of queue)	Deleted	Deleted	Deleted
Temporary dynamic queue (call not issued by creator of queue)	Retained	Not valid	Not valid
Distribution list	Retained	Not valid	Not valid
Managed subscription destination	Retained	Not valid	Not valid
Distribution list (subscription has been removed)	Messages deleted; queue deleted	Not valid	Not valid

Subscription closure options

These options control whether durable subscriptions are removed when the handle is closed, and whether publications still waiting to be read by the application are cleaned up. These options are only valid for use with an object handle returned in the *HSUB* parameter of an MQSUB call.

COKPSB

The handle to the subscription is closed but the subscription made is kept. Publications will continue to be sent to the destination specified in the subscription. This option is only valid if the subscription was made with the option SODUR. COKPSB is the default if the subscription is durable

CORMSB

The subscription is removed and the handle to the subscription is closed.

The *Hobj* parameter of the MQSUB call is not invalidated by closure of the *Hsub* parameter and may continue to be used for MQGET or MQCB to receive the remaining publications. When the *Hobj* parameter of the MQSUB call is also closed, if it was a managed destination any unretrieved publications will be removed.

CORMSB is the default if the subscription is non-durable.

These subscription closure options are summarized in the following tables:

To close a durable subscription handle but leave the subscription around, use the following subscription closure options:

Task	Subscription closure option
Keep publications on an MQOPENed handle	COKPSB
Remove publications on an MQOPENed handle	Action not allowed
Keep publications on a handle with SOMAN	COKPSB
Remove publications on a handle with SOMAN	Action not allowed

To unsubscribe, either by closing a durable subscription handle and unsubscribing it or closing a non-durable subscription handle, use the following subscription closure options:

Task	Subscription closure option
Keep publications on an MQOPENed handle	CORMSB
Remove publications on an MQOPENed handle	Action not allowed
Keep publications on a handle with SOMAN	CORMSB
Remove publications on a handle with SOMAN	COPGSB

Read ahead options

The following options control what happens to non-persistent messages which have been sent to the client before an application requested them and have not yet been consumed by the application. These messages are stored in the client read ahead buffer waiting to be requested by the application and can either be discarded or consumed from the queue before the MQCLOSE is completed.

COIMM

The object is closed immediately and any messages which have been sent to the client before an application requested them are discarded and are not available to be consumed by any application. This is the default value.

COQSC

A request to close the object is made, but if any messages which have been sent to the client before an application requested them, still reside in the client read ahead buffer, the MQCLOSE call will return with a warning code of RC2458, and the object handle will remain valid.

The application can then continue to use the object handle to retrieve messages until no more are available, and then close the object again. No more messages will be sent to the client ahead of an application requesting then, read ahead is now turned off.

Applications are advised to use COQSC rather than trying to reach a point where there are no more messages in the client read ahead buffer, since a message could arrive between the last MQGET call and the following MQCLOSE which would be discarded if COIMM was used.

If an MQCLOSE with COQSC is issued from within an asynchronous callback function, the same behavior of reading ahead messages applies. If the warning code RC2458 is returned, then the callback function will be called at least one more time. When the last remaining message that was read ahead has been passed to the callback function the CBCFLG field is set to CBCFBE.

Default option

If you require none of the options described previously, you can use the following option:

CONONE

No optional close processing required.

This *must* be specified for:

- Objects other than queues
- Predefined queues
- Temporary dynamic queues (but only in those cases where *HOBJ* is *not* the handle returned by the MQOPEN call that created the queue).
- Distribution lists

In all of the previous cases, the object is retained and not deleted.

If this option is specified for a temporary dynamic queue:

- The queue is deleted, if it was created by the MQOPEN call that returned *HOBJ* ; any messages that are on the queue are purged.
- In all other cases the queue (and any messages on it) are retained.

If this option is specified for a permanent dynamic queue, the queue is retained and not deleted.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2241

(2241, X'8C1') Message group not complete.

RC2242

(2242, X'8C2') Logical message not complete.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2035

(2035, X'7F3') Not authorized for access.

RC2101

(2101, X'835') Object damaged.

RC2045

(2045, X'7FD') Option not valid for object type.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2055

(2055, X'807') Queue contains one or more messages or uncommitted put or get requests.

RC2102

(2102, X'836') Insufficient system resources available.

RC2063

(2063, X'80F') Security error occurred.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..  
C          CALLP      MQCLOSE(HCONN : HOBJ : OPTS :  
C                                CMPCOD : REASON)
```


The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..  
DMQCLOSE          PR          EXTPROC('MQCLOSE')  
D* Connection handle  
D HCONN              10I 0 VALUE  
D* Object handle  
D HOBJ              10I 0  
D* Options that control the action of MQCLOSE  
D OPTS              10I 0 VALUE  
D* Completion code  
D CMPCOD            10I 0  
D* Reason code qualifying CMPCOD  
D REASON            10I 0
```

MQCMIT - Commit changes

The MQCMIT call indicates to the queue manager that the application has reached a syncpoint, and that all of the message gets and puts that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work are made available to other applications; messages retrieved as part of a unit of work are deleted.

- On IBM i, this call is not supported for applications running in compatibility mode.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

- [“Syntax” on page 1230](#)
- [“Usage notes” on page 1230](#)
- [“Parameters” on page 1231](#)
- [“RPG Declaration” on page 1232](#)

Syntax

MQCMIT (*HCONN*, *COMCOD*, *REASON*)

Usage notes

Consider these usage notes when using MQCMIT.

1. This call can be used only when the queue manager itself coordinates the unit of work. This is a local unit of work, where the changes affect only IBM MQ resources.
2. In environments where the queue manager does not coordinate the unit of work, the appropriate commit call must be used instead of MQCMIT. The environment may also support an implicit commit caused by the application terminating normally.
 - On IBM i, this call can be used for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
3. If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager” on page 1246](#) for further details.
4. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

When a unit of work is committed, the queue manager retains the group and segment information, and the application can continue putting or getting messages in the current message group or logical message.

Retaining the group and segment information when a unit of work is committed allows the application to spread a large message group or large logical message consisting of many segments across several units of work. Using several units of work might be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point if a system failure occurs. For details of how to restart at the correct point after a system failure, see the PMLOGO option described in [“MQPMO -](#)

Put-message options” on page 1131, and the GMLOGO option described in “MQGMO - Get-message options” on page 1035.

The remaining usage notes apply only when the queue manager coordinates the units of work:

1. A unit of work has the same scope as a connection handle. This means that all IBM MQ calls which affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *HCONN* parameter described in MQCONN for information about the scope of connection handles.
2. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
3. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but which never issues a commit or back-out call, can cause queues to fill up with messages that are not available to other applications. To guard against this possibility, the administrator should set the *MaxUncommittedMsgs* queue manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.

Parameters

The MQCMIT call has the following parameters:

HCONN (10-digit signed integer) - input

Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

COMCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *COMCOD*.

If *COMCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *COMCOD* is CCWARN:

RC2003

(2003, X'7D3') Unit of work backed out.

RC2124

(2124, X'84C') Result of commit operation is pending.

If *COMCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2101

(2101, X'835') Object damaged.

RC2123

(2123, X'84B') Result of commit or back-out operation is mixed.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQCMIT(HCONN : COMCOD : REASON)

```

The prototype definition for the call is:

```

D*.1.....2.....3.....4.....5.....6.....7..
DMQCMIT          PR          EXTPROC('MQCMIT')
D* Connection handle
D HCONN          10I 0 VALUE
D* Completion code
D COMCOD          10I 0
D* Reason code qualifying COMCOD
D REASON          10I 0


```

MQCONN - Connect queue manager

The MQCONN call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent message queuing calls.

- On IBM i, applications running in compatibility mode do not have to issue this call. These applications are connected automatically to the queue manager when they issue the first MQOPEN call. However, the MQCONN and MQDISC calls are still accepted from IBM i applications.

Other applications (that is, applications not running in compatibility mode) must use the MQCONN or MQCONNEX call to connect to the queue manager, and the MQDISC call to disconnect from the queue manager. Consider using this style of programming.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

On IBM MQ for Windows, UNIX, and IBM i, each thread in an application can connect to different queue managers. On other systems, all concurrent connections within a process must be to the same queue manager.

- [“Syntax” on page 1233](#)
- [“Usage notes” on page 1233](#)
- [“Parameters” on page 1234](#)
- [“RPG Declaration” on page 1236](#)

Syntax

MQCONN (QMNAME, HCONN, CMPCOD, REASON)

Usage notes

1. The queue manager to which connection is made using the MQCONN call is called the *local queue manager*.
2. Queues that are owned by the local queue manager appear to the application as local queues. It is possible to put messages on and get messages from these queues.

Shared queues that are owned by the queue-sharing group to which the local queue manager belongs appear to the application as local queues. It is possible to put messages on and get messages from these queues.

Queues that are owned by remote queue managers appear as remote queues. It is possible to put messages on these queues, but not possible to get messages from these queues.
3. If the queue manager fails while an application is running, the application must issue the MQCONN call again in order to obtain a new connection handle to use on subsequent IBM MQ calls. The application can issue the MQCONN call periodically until the call succeeds.

If an application is not sure whether it is connected to the queue manager, the application can safely issue an MQCONN call in order to obtain a connection handle. If the application is already connected, the handle returned is the same as that returned by the previous MQCONN call, but with completion code CCWARN and reason code RC2002.

4. When the application has finished using IBM MQ calls, the application should use the MQDISC call to disconnect from the queue manager.
5. On IBM i, applications written for releases earlier than MQSeries V5.1 of the queue manager can run without the need for recompilation.
6. This is a *compatibility mode*. This mode of operation provides a compatible runtime environment for applications written using the dynamic linkage. It comprises the following:

- The service program AMQZSTUB residing in the library QMQM.


AMQZSTUB provides the same public interface as previous releases, and has the same signature. This service program can be used to access the MQI through bound procedure calls.

- The program QMQM residing in the library QMQM.

QMQM provides a means of accessing the MQI through dynamic program calls.

- Programs MQCLOSE, MQCONN, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET residing in the library QMQM.

These programs also provide a means of accessing the MQI through dynamic program calls, but with a parameter list that corresponds to the standard descriptions of the IBM MQ calls.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

These three interfaces do not include capabilities that were introduced in version 5.1. For example, the MQBACK, MQCMIT, and MQCONNX calls are not supported. The support provided by these interfaces is for single-threaded applications only.

Support for the static bound IBM MQ calls in single-threaded applications, and for all IBM MQ calls in multi-threaded applications, is provided through the service programs LIBMQM and LIBMQM_R.

7. On IBM i, programs that end abnormally are not automatically disconnected from the queue manager. Therefore applications should be written to allow for the possibility of the MQCONN or MQCONNX call returning completion code CCWARN and reason code RC2002. The connection handle returned in this situation can be used as normal.

Parameters

The MQCONN call has the following parameters:

QMNAME (48-byte character string) - input

Name of queue manager.

This is the name of the queue manager to which the application wants to connect. The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but might contain trailing blanks. A null character can be used to indicate the end of significant data in the name; the null and any characters following it are treated as blanks. The following restrictions apply in the environments indicated:

- On IBM i, names containing lowercase characters, forward slash, or percent must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified in the *QMNAME* parameter.

If the name consists entirely of blanks, the name of the *default* queue manager is used.

The name specified for *QMNAME* must be the name of a *connectable* queue manager.

Queue-sharing groups: On systems where several queue managers exist and are configured to form a queue-sharing group, the name of the queue-sharing group can be specified for *QMNAME* in place of the name of a queue manager. This allows the application to connect to *any* queue manager that is available in the queue-sharing group. The system can also be configured so that a blank *QMNAME* causes connection to the queue-sharing group instead of to the default queue manager.

If *QMNAME* specifies the name of the queue-sharing group, but there is also a queue manager with that name on the system, connection is made to the latter in preference to the former. Only if that connection fails is connection to one of the queue managers in the queue-sharing group attempted.

If the connection is successful, the handle returned by the MQCONN or MQCONNX call can be used to access *all* of the resources (both shared and nonshared) that belong to the particular queue manager to which connection has been made. Access to these resources is subject to the typical authorization controls.

If the application issues two MQCONN or MQCONNX calls in order to establish concurrent connections, and one or both calls specifies the name of the queue-sharing group, the second call may return completion code CCWARN and reason code RC2002. This occurs when the second call connects to the same queue manager as the first call.

Queue-sharing groups are supported only on z/OS. Connection to a queue-sharing group is supported only in the batch, RRS batch, and TSO environments.

IBM MQ client applications: For IBM MQ MQI client applications, a connection is attempted for each client-connection channel definition with the specified queue manager name, until one is successful. The queue manager, however, must have the same name as the specified name. If an all-blank name is specified, each client-connection channel with an all-blank queue manager name is tried until one is successful; in this case there is no check against the actual name of the queue manager.

IBM MQ client queue manager groups: If the specified name starts with an asterisk (*), the actual queue manager to which connection is made may have a name that is different from that specified by the application. The specified name (without the asterisk) defines a *group* of queue managers that are eligible for connection. The implementation selects one from the group by trying each one in turn, in alphabetic order, until one is found to which a connection can be made. If none of the queue managers in the group is available for connection, the call fails. Each queue manager is tried once

only. If an asterisk alone is specified for the name, an implementation-defined default queue manager group is used.

Queue-manager groups are supported only for applications running in an MQ-client environment; the call fails if a non-client application specifies a queue manager name beginning with an asterisk. A group is defined by providing several client connection channel definitions with the same queue manager name (the specified name without the asterisk), to communicate with each of the queue managers in the group. The default group is defined by providing one or more client connection channel definitions, each with a blank queue manager name (specifying an all-blank name therefore has the same effect as specifying a single asterisk for the name for a client application).

After connecting to one queue manager of a group, an application can specify blanks in the typical way in the queue manager name fields in the message and object descriptors to mean the name of the queue manager to which the application has actually connected (the *local queue manager*). If the application needs to know this name, the MQINQ call can be issued to inquire the *QMGrName* queue manager attribute.

Prefixing an asterisk to the connection name implies that the application is not dependent on connecting to a particular queue manager in the group. Suitable applications would be:

- Applications that put messages but do not get messages.
- Applications that put request messages and then get the reply messages from a *temporary dynamic* queue.

Unsuitable applications would be those that need to get messages from a particular queue at a particular queue manager; such applications should not prefix the name with an asterisk.

Note that if an asterisk is specified, the maximum length of the remainder of the name is 47 characters.

The length of this parameter is given by LNQMNL.

HCONN (10-digit signed integer) - output

Connection handle.

This handle represents the connection to the queue manager. It must be specified on all subsequent message queuing calls issued by the application. It ceases to be valid when the MQDISC call is issued, or when the unit of processing that defines the scope of the handle terminates.

The scope of the handle is restricted to the smallest unit of parallel processing supported by the platform on which the application is running; the handle is not valid outside the unit of parallel processing from which the MQCONN call was issued.

- On IBM i, the scope of the handle is the job issuing the call.

On IBM i for applications running in compatibility mode, the value returned is:

HCDEFH

Default connection handle.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2002

(2002, X'7D2') Application already connected.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2267

(2267, X'8DB') Unable to load cluster workload exit.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2035

(2035, X'7F3') Not authorized for access.

RC2137

(2137, X'859') Object not opened successfully.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2161

(2161, X'871') Queue manager quiescing.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2063

(2063, X'80F') Security error occurred.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```
C*..1.....2.....3.....4.....5.....6.....7..  
C                                CALLP      MQCONN(QMNAME : HCONN : CMPCOD :  
C                                REASON)
```

The prototype definition for the call is:


```
D*..1.....2.....3.....4.....5.....6.....7..  
DMQCONN      PR      EXTPROC('MQCONN')  
D* Name of queue manager  
D QMNAME      48A  
D* Connection handle  
D HCONN      10I 0  
D* Completion code  
D CMPCOD      10I 0  
D* Reason code qualifying CMPCOD  
D REASON      10I 0
```

MQCONN - Connect queue manager (extended)

The MQCONN call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent IBM MQ calls.

The MQCONN call is like the MQCONN call, except that MQCONN allows options to be specified to control the way that the call works.

- On IBM i, this call is not supported for applications running in compatibility mode.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

On IBM MQ for Windows, UNIX, and IBM i, each thread in an application can connect to different queue managers. On other systems, all concurrent connections within a process must be to the same queue manager.

- [“Syntax” on page 1237](#)
- [“Parameters” on page 1237](#)
- [“RPG Declaration” on page 1238](#)

Syntax

MQCONN (*QMNAME*, *CNOPT*, *HCONN*, *CMPCOD*, *REASON*)

Parameters

The MQCONN call has the following parameters:

QMNAME (48-byte character string) - input

Name of queue manager.

See the *QMNAME* parameter described in [“MQCONN - Connect queue manager” on page 1232](#) for details.

CNOPT (MQCNO) - input/output

Options that control the action of MQCONN.

See [“MQCNO - Connect options” on page 1009](#) for details.

HCONN (10-digit signed integer) - output

Connection handle.

See the *HCONN* parameter described in [“MQCONN - Connect queue manager” on page 1232](#) for details.

CMPCOD (10-digit signed integer) - output

Completion code.

See the *CMPCOD* parameter described in [“MQCONN - Connect queue manager” on page 1232](#) for details.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

See the *REASON* parameter described in [“MQCONN - Connect queue manager” on page 1232](#) for details of possible reason codes.

The following additional reason codes can be returned by the MQCONN call:

If *CMPCOD* is CCFAIL:

RC2278

(2278, X'8E6') Client connection fields not valid.

RC2139

(2139, X'85B') Connect-options structure not valid.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQCONN(QMNAME : HCONN : CMPCOD :
C                      REASON)

```

The prototype definition for the call is:

```

D*.1.....2.....3.....4.....5.....6.....7..
DMQCONN      PR          EXTPROC('MQCONN')
D* Name of queue manager
D QMNAME          48A
D* Options that control the action of MQCONN
D HCONN          224A
D* Connection handle
D HCONN          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0

```

MQCRTMH - Create message handle

The MQCRTMH call returns a message handle.

An application can use it on subsequent message queuing calls:

- Use the [MQSETMP](#) call to set a property of the message handle.
- Use the [MQINQMP](#) call to inquire on the value of a property of the message handle.
- Use the [MQDLTMP](#) call to delete a property of the message handle.

The message handle can be used on the MQPUT and MQPUT1 calls to associate the properties of the message handle with the properties of the message being put. Similarly, by specifying a message handle on the MQGET call, the properties of the message being retrieved can be accessed by using the message handle when the MQGET call completes.

Use [MQDLTMH](#) to delete the message handle.

- [“Syntax” on page 1238](#)
- [“Parameters” on page 1238](#)
- [“RPG Declaration” on page 1240](#)

Syntax

MQCRTMH (*Hconn*, *CrtMsgHOpts*, *Hmsg*, *CompCode*, *Reason*)

Parameters

The MQCRTMH call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNX call. If the connection to the queue manager ceases to be valid and no IBM MQ call is operating on the message handle, [MQDLTMH](#) is implicitly called to delete the message.

Alternatively, you can specify the following value:

HCUNAS

The connection handle does not represent a connection to any particular queue manager.


When this value is used, the message handle must be deleted with an explicit call to [MQDLTMH](#) in order to release any storage allocated to it; IBM MQ never implicitly deletes the message handle.

There must be at least one valid connection to a queue manager established on the thread creating the message handle, otherwise the call fails with RC2018.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and you can specify the following value for *HCONN* :

HCDEFH

Default connection handle

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

CRTOPT (MQCMHO) - input

The options that control the action of MQCRTMH. See [MQCMHO](#) for details.

HMSG (20-digit signed integer) - output

On output a message handle is returned that can be used to set, inquire, and delete properties of the message handle. Initially the message handle contains no properties.

A message handle also has an associated message descriptor. Initially this message descriptor contains the default values. The values of the associated message descriptor fields can be set and inquired by using the MQSETMP and MQINQMP calls. The MQDLTMP call resets a field of the message descriptor back to its default value.

If the *HCONN* parameter is specified as the value HCUNAS then the returned message handle can be used on MQGET, MQPUT, or MQPUT1 calls with any connection within the unit of processing, but can be in use by only one IBM MQ call at a time. If the handle is in use when a second IBM MQ call attempts to use the same message handle, the second IBM MQ call fails with reason code RC2499.

If the *HCONN* parameter is not HCUNAS then the returned message handle can be used only on the specified connection.

The same *HCONN* parameter value must be used on the subsequent MQI calls where this message handle is used:

- MQDLTMH
- MQSETMP
- MQINQMP
- MQDLTMP
- MQMHBUFF
- MQBUFMH

The returned message handle ceases to be valid when the MQDLTMH call is issued for the message handle, or when the unit of processing that defines the scope of the handle terminates. MQDLTMH is called implicitly if a specific connection is supplied when the message handle is created and the connection to the queue manager ceases to be valid, for example, if MQDBC is called.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2461

(2461, X'099D') Create message handle options structure not valid.

RC2273

(2273, X'7D9') Connection to queue manager lost.

RC2017

(2017, X'07E1') No more handles available.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2460

(2460, X'099C') Message handle pointer not valid.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See [“Return codes for IBM i \(ILE RPG\)” on page 1381](#) for more details.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQCRTMH(HCONN : CRTOPT : HMSG :
                                           CMPCOD : REASON)
```

The prototype definition for the call is:

```
DMQCRTMH      PR              EXTPROC('MQCRTMH')
D* Connection handle
D HCONN              10I 0 VALUE
D* Options that control the action of MQCRTMH
D CRTOPT              12A
D* Message handle
D HMSG              20I 0
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CompCode
D REASON              10I 0
```

MQCTL - Control callback

The MQCTL call performs controlling actions on the object handles opened for a connection.

- [“Syntax” on page 1241](#)
- [“Usage notes” on page 1241](#)
- [“Parameters” on page 1241](#)
- [“RPG Declaration” on page 1246](#)

Syntax

MQCTL (*Hconn*, *Operation*, *ControlOpts*, *CompCode*, *Reason*)

Usage notes

1. Callback routines must check the responses from all services they invoke, and if the routine detects a condition that cannot be resolved, it must issue an MQCB(CBREG) command to prevent repeated calls to the callback routine.

Parameters

The MQCTL call has the following parameters:


HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and you can specify the following special value for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

OPERATN (10-digit signed integer) - input

The operation being processed on the callback defined for the specified object handle. You must specify one, and one only, of the following options:

CTLSR

Start the consuming of messages for all defined message consumer functions for the specified connection handle.

Callbacks run on a thread started by the system, which is different from any of the application threads.

This operation gives control of the provided connection handle to system. The only MQI calls which can be issued by a thread other than the consumer thread are:

- MQCTL with Operation CTLSP
- MQCTL with Operation CTLSU
- MQDISC - This performs MQCTL with Operation CTLSP before disconnection the HConn.

RC2500 is returned if an IBM MQ API call is issued while the connection handle is started, and the call does not originate from a message consumer function.

If a connection fails, this stops the conversation as soon as possible. It is possible, therefore, for an IBM MQ API call being issued on the main thread to receive the return code RC2500 for a while, followed by the return code RC2009 when the connection reverts to the stopped state.

This can be issued in a consumer function. For the same connection as the callback routine, its only purpose is to cancel a previously issued CTLSP operation.

This option is not supported if the application is bound with a nonthreaded IBM MQ library.

CTLSW

Start the consuming of messages for all defined message consumer functions for the specified connection handle.

Message consumers run on the same thread and control is not returned to the caller of MQCTL until:

- Released by the use of the MQCTL CTLSP or CTLSU operations, or
- All consumer routines have been deregistered or suspended.

If all consumers are deregistered or suspended, an implicit CTLSP operation is issued.

This option cannot be used from within a callback routine, either for the current connection handle or any other connection handle. If the call is attempted it returns with RC2012.

If, at any time during a CTLSW operation there are no registered, non-suspended consumers the call fails with a reason code of RC2446.

If, during a CTLSW operation, the connection is suspended, the MQCTL call returns a warning reason code of RC2521; the connection remains 'started'.

The application can choose to issue CTLSP or CTLRE. In this instance, the CTLRE operation blocks.

This option is not supported in a single threaded client.

CTLSP

Stop the consuming of messages, and wait for all consumers to complete their operations before this option completes. This operation releases the connection handle.

If issued from within a callback routine, this option does not take effect until the routine exits. No more message consumer routines are called after the consumer routines for messages already read have completed, and after stop calls (if requested) to callback routines have been made.

If issued outside a callback routine, control does not return to the caller until the consumer routines for messages already read have completed, and after stop calls (if requested) to callbacks have been made. The callbacks themselves, however, remain registered.

This function has no effect on read ahead messages. You must ensure that consumers run MQCLOSE(COQSC), from within the callback function, to determine whether there are any further messages available to be delivered.

CTLSU

Pause the consuming of messages. This operation releases the connection handle.

This does not affect the reading ahead of messages for the application. If you intend to stop consuming messages for a long period, consider closing the queue and reopening it when consumption must continue.

If issued from within a callback routine, it does not take effect until the routine exits. No more message consumer routines will be called after the current routine exits.

If issued outside a callback, control does not return to the caller until the current consumer routine has completed and no more are called.

CTLRE

Resume the consuming of messages.

This option is normally issued from the main application thread, but it can also be used from within a callback routine to cancel an earlier suspension request issued in the same routine.

If CTLRE is used to resume a CTLSW, then the operation blocks.

PCTLOP (MQCTLO) - input

Options that control the action of MQCTL

See [MQCTLO](#) for details of the structure.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The following reason codes are the ones that the queue manager can return for the *Reason* parameter.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2133

(2133, X'855') Unable to load data conversion services modules.

RC2204

(2204, X'89C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2374

(2374, X'946') API exit failed.

RC2183

(2183, X'887') Unable to load API exit.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2487

(2487, X'9B7') Unable to call the callback routine

RC2448

(2448, X'990') Unable to Deregister, Suspend, or Resume because there is no registered callback

RC2486

(2486, X'9B6') Either, both CallbackFunction and CallbackName have been specified on a CBREG call, or either one of CallbackFunction or CallbackName has been specified but does not match the currently registered callback function.

RC2483

(2483, X'9B3') Incorrect CallBackType field.

RC2219

(2219, X'8AB') MQI call entered before previous call complete.

RC2444

(2444, X'98C') Option block is incorrect.

RC2484
(2484, X'9B4') Incorrect MQCBD options field.

RC2140
(2140, X'85C') Wait request rejected by CICS.

RC2009
(2009, X'7D9') Connection to queue manager lost.

RC2217
(2217, X'8A9') Not authorized for connection.

RC2202
(2202, X'89A') Connection quiescing.

RC2203
(2203, X'89B') Connection shutting down.

RC2207
(2207, X'89F') Correlation-identifier error.

RC2016
(2016, X'7E0') Gets inhibited for the queue.

RC2351
(2351, X'92F') Global units of work conflict.

RC2186
(2186, X'88A') Get-message options structure not valid.

RC2353
(2353, X'931') Handle in use for global unit of work.

RC2018
(2018, X'7E2') Connection handle not valid.

RC2019
(2019, X'7E3') Object handle not valid.

RC2259
(2259, X'8D3') Inconsistent browse specification.

RC2245
(2245, X'8C5') Inconsistent unit-of-work specification.

RC2246
(2246, X'8C6') Message under cursor not valid for retrieval.

RC2352
(2352, X'930') Global unit of work conflicts with local unit of work.

RC2247
(2247, X'8C7') Match options not valid.

RC2485
(2485, X'9B5') Incorrect MaxMsgLength field

RC2026
(2026, X'7EA') Message descriptor not valid.

RC2497
(2497, X'9C1')The specified function entry point was not be found in the module.

RC2496
(2496, X'9C0') Module is found but is of the wrong type (32-bit or 64-bit) or is not a valid dll.

RC2495
(2495, X'9BF') Module not found in the search path or not authorized to load.

RC2206
(2206, X'89E') Message-identifier error.

RC2250
(2250, X'8CA') Message sequence number not valid.

RC2331
(2331, X'91B') Use of message token not valid.

RC2036
(2036, X'7F4') Queue not open for browse.

RC2037
(2037, X'7F5') Queue not open for input.

RC2041
(2041, X'7F9') Object definition changed since opened.

RC2101
(2101, X'835') Object damaged.

RC2488
(2488, X'9B8') Incorrect Operation code on API Call

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2193
(2193, X'891') Error accessing page-set data set.

RC2052
(2052, X'804') Queue has been deleted.

RC2394
(2394, X'95A') Queue has wrong index type.

RC2058
(2058, X'80A') Queue manager name not valid or not known.

RC2059
(2059, X'80B') Queue manager not available for connection.

RC2161
(2161, X'871') Queue manager quiescing.

RC2162
(2162, X'872') Queue manager shutting down.

RC2102
(2102, X'836') Insufficient system resources available.

RC2069
(2069, X'815') Signal outstanding for this handle.

RC2071
(2071, X'817') Insufficient storage available.

RC2109
(2109, X'83D') Call suppressed by exit program.

RC2072
(2072, X'818') Syncpoint support not available.

RC2195
(2195, X'893') Unexpected error occurred.

RC2354
(2354, X'932') Enlistment in global unit of work failed.

RC2355
(2355, X'933') Mixture of unit-of-work calls not supported.

RC2255
(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2090
(2090, X'82A') Wait interval in MQGMO not valid.

RC2256
(2256, X'8D0') Wrong version of MQGMO supplied.

RC2257

(2257, X'8D1') Wrong version of MQMD supplied.

RC2298

(2298, X'8FA') The function requested is not available in the current environment.

RPG Declaration

```

C*..1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQCTL(HCONN : OPERATN : PCTLOP :
                                           CMPCOD : REASON)

```

The prototype definition for the call is:

```


DMQCTL          PR                EXTPROC('MQCTL')
D* Connection handle
D HCONN          10I 0 VALUE
D* Operation
D OPERATN        10I 0 VALUE
D* Control options
D PCTLOP          32A
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CompCode
D REASON          10I 0

```

MQDISC - Disconnect queue manager

The MQDISC call breaks the connection between the queue manager and the application program, and is the inverse of the MQCONN or MQCONNX call.

- On IBM i, applications running in compatibility mode do not need to issue this call. See [“MQCONN - Connect queue manager” on page 1232](#) for more information.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

- [“Syntax” on page 1246](#)
- [“Usage notes” on page 1246](#)
- [“Parameters” on page 1247](#)
- [“RPG Declaration” on page 1248](#)

Syntax

MQDISC (*HCONN*, *CMPCOD*, *REASON*)

Usage notes

1. If an MQDISC call is issued when the application still has objects open, those objects are closed by the queue manager, with the close options set to CONONE.
2. If the application ends with uncommitted changes in a unit of work, the disposition of those changes depends on how the application ends:
 - a. If the application issues the MQDISC call before ending:
 - For a queue manager coordinated unit of work, the queue manager issues the MQCMIT call on behalf of the application. The unit of work is committed if possible, and backed out if not.
 - For an externally coordinated unit of work, there is no change in the status of the unit of work; however, the queue manager will indicate that the unit of work should be committed, when asked by the unit-of-work coordinator.

- b. If the application ends normally but without issuing the MQDISC call, the unit of work is backed out.
 - c. If the application ends *abnormally* without issuing the MQDISC call, the unit of work is backed out.
3. On IBM i, applications running in compatibility mode do not have to issue this call; see the MQCONN call for more details.

Parameters

The MQDISC call has the following parameters:

HCONN (10-digit signed integer) - input/output

Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

On successful completion of the call, the queue manager sets *HCONN* to a value that is not a valid handle for the environment. This value is:

HCUNUH

Unusable connection handle.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQDISC(HCONN : CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*.1.....2.....3.....4.....5.....6.....7..
DMQDISC      PR      EXTPROC('MQDISC')
D* Connection handle
D HCONN      10I 0
D* Completion code
D CMPCOD      10I 0
D* Reason code qualifying CMPCOD
D REASON      10I 0

```

MQDLTMH - Delete message handle

The MQDLTMH call deletes a message handle and is the inverse of the MQCRTMH call.

- [“Syntax” on page 1248](#)
- [“Usage notes” on page 1248](#)
- [“Parameters” on page 1250](#)
- [“RPG Declaration” on page 1251](#)

Syntax

MQDLTMH ((*Hconn*, *Hmsg*, *DltMsgHOpts*, *CompCode*, *Reason*))

Usage notes

1. You can use this call only when the queue manager itself coordinates the unit of work. This can be:
 - A local unit of work, where the changes affect only IBM MQ resources.
 - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting IBM MQ resources.

For further details about local and global units of work, see [“MQBEGIN - Begin unit of work” on page 1209](#).

2. In environments where the queue manager does not coordinate the unit of work, use the appropriate back-out call instead of MQBACK. The environment might also support an implicit back out caused by the application terminating abnormally.
 - On z/OS, use the following calls:
 - Batch programs (including IMS batch DL/I programs) can use the MQBACK call if the unit of work affects only IBM MQ resources. However, if the unit of work affects both IBM MQ resources and resources belonging to other resource managers (for example, Db2), use the SRRBACK call provided by the z/OS Recoverable Resource Service (RRS). The SRRBACK call backs out changes to resources belonging to the resource managers that have been enabled for RRS coordination.

- CICS applications must use the EXEC CICS SYNCPOINT ROLLBACK command to back out the unit of work. Do not use the MQBACK call for CICS applications.
 - IMS applications (other than batch DL/I programs) must use IMS calls such as ROLB to back out the unit of work. Do not use the MQBACK call for IMS applications (other than batch DL/I programs).
 - On IBM i, use this call for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
3. If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager”](#) on page 1246 for further details.
 4. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

The queue manager keeps three sets of group and segment information, one set for each of the following:

- The last successful MQPUT call (this can be part of a unit of work).
- The last successful MQGET call that removed a message from the queue (this can be part of a unit of work).
- The last successful MQGET call that browsed a message on the queue (this cannot be part of a unit of work).

If the application puts or gets the messages as part of a unit of work, and the application then backs out the unit of work, the group and segment information is restored to the value that it had previously:

- The information associated with the MQPUT call is restored to the value that it had before the first successful MQPUT call for that queue handle in the current unit of work.
- The information associated with the MQGET call is restored to the value that it had before the first successful MQGET call for that queue handle in the current unit of work.

Queues that were updated by the application after the unit of work started, but outside the scope of the unit of work, do not have their group and segment information restored if the unit of work is backed out.

Restoring the group and segment information to its previous value when a unit of work is backed out allows the application to spread a large message group or large logical message consisting of many segments across several units of work, and to restart at the correct point in the message group or logical message if one of the units of work fails. Using several units of work might be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point if that a system failure occurs.

For details of how to restart at the correct point after a system failure, see the PMLOGO option described in [PMOPT \(10 digit signed integer\)](#), and the GMLOGO option described in [GMOPT \(10 digit signed integer\)](#).

The remaining usage notes apply only when the queue manager coordinates the units of work:

5. A unit of work has the same scope as a connection handle. All IBM MQ calls that affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See [HCONN \(10 digit signed integer\) - output](#) for information about the scope of connection handles.

6. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
7. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but that never issues a commit or backout call, can fill queues with messages that are not available to other applications. To guard against this possibility, the administrator must set the *MaxUncommittedMsgs* queue-manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.

Parameters

The MQDLTMH call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager.

The value must match the connection handle that was used to create the message handle specified in the *HMSG* parameter.

If the message handle was created using HCUNAS then a valid connection must be established on the thread deleting the message handle, otherwise the call fails with RC2009 .

HMSG (20-digit signed integer) - input/output

This is the message handle to be deleted. The value was returned by a previous MQCRTMH call.

On successful completion of the call, the handle is set to an invalid value for the environment. This value is:

HMUNUH

Unusable message handle.

The message handle cannot be deleted if another IBM MQ call is in progress that was passed the same message handle.

DLTOPT (MQDMHO) - input

See [MQDMHO](#) for details.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2009

(2009, X'07D9') Connection to queue manager lost.

RC2462

(2462, X'099E') Delete message handle options structure not valid.

RC2460

(2460, X'099C') Message handle pointer not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See [“Return codes for IBM i \(ILE RPG\)”](#) on page 1381 for more details.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQDLTMH(HCONN : HMSG : DLTOPT :
                                           CMPCOD : REASON)

```

The prototype definition for the call is:

```

DMQDLTMH          PR              EXTPROC('MQDLTMH')
D* Connection handle
D HCONN              10I 0 VALUE
D* Message handle
D HMSG              20I 0
D* Options that control the action of MQDLTMH
D DLTOPT            12A
D* Completion code
D CMPCOD            10I 0
D* Reason code qualifying CompCode
D REASON            10I 0

```

MQDLTMP - Delete message property

The MQDLTMP call deletes a property from a message handle and is the inverse of the MQSETMP call.

- [“Syntax”](#) on page 1251
- [“Parameters”](#) on page 1251
- [“RPG Declaration”](#) on page 1253

Syntax

MQDLTMP (*Hconn*, *Hmsg*, *DltPropOpts*, *Name*, *CompCode*, *Reason*)

Parameters

The MQDLTMP call has the following parameters:

HCONN (10-digit signed integer) - Input

This handle represents the connection to the queue manager. The value must match the connection handle that was used to create the message handle specified in the *HMSG* parameter.

If the message handle was created using HCUNAS then a valid connection must be established on the thread deleting the message handle otherwise the call fails with RC2009.

HMSG (20-digit signed integer) - input

This is the message handle containing the property to be deleted. The value was returned by a previous MQCRTMH call.

DLTOPT (MQDMPO) - Input

See the [MQDMPO](#) data type for details.

PRNAME (MQCHARV) - input

The name of the property to delete. See [Property names](#) for further information about property names.

Wildcards are not allowed in the property name.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2471

(2471, X'09A7') Property not available.

RC2421

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'0852') Unable to load adapter service module.

RC2157

(2157, X'086D') Primary and home ASIDs differ.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2009

(2009, X'07D9') Connection to queue manager lost.

RC2481

(2481, X'09B1') Delete message property options structure not valid.

RC2460

(2460, X'099C') Message handle not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2442

(2442, X'098A') Invalid property name.

RC2111

(2111, X'083F') Property name coded character set identifier not valid.

RC2195

(2195, X'0893') Unexpected error occurred.

For more information about these codes, see [API completion and reason codes](#).

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQDLTMP(HCONN : HMSG : DLTOPT :
                                           PRNAME : CMPCOD : REASON)

```

The prototype definition for the call is:

```

MQDLTMP          PR          EXTPROC('MQDLTMP')
D* Connection handle
D HCONN          10I 0 VALUE
D* Message handle
D HMSG          20I 0 VALUE
D* Options that control the action of MQDLTMP
D DLTOPT        12A
D* Property name
D PRNAME        32A
D* Completion code
D CMPCOD        10I 0
D* Reason code qualifying CompCode
D REASON        10I 0

```

MQGET - Get message

The MQGET call retrieves a message from a local queue that has been opened by using the MQOPEN call.

- [“Syntax” on page 1253](#)
- [“Usage notes” on page 1253](#)
- [“Parameters” on page 1256](#)
- [“RPG Declaration” on page 1261](#)

Syntax

MQGET (*HCONN*, *HOBJ*, *MSGDSC*, *GMO*, *BUFLEN*, *BUFFER*, *DATLEN*, *CMPCOD*, *REASON*)

Usage notes

1. The message retrieved is normally deleted from the queue. This deletion can occur as part of the MQGET call itself, or as part of a syncpoint. Message deletion does not occur if a GMBRWF or GMBRWN option is specified on the *GMO* parameter (see the *GMOPT* field described in [“MQGMO - Get-message options” on page 1035](#)).
2. If the GMLK option is specified with one of the browse options, the browsed message is locked so that it is visible only to this handle.

If the GMUNLK option is specified, a previously locked message is unlocked. No message is retrieved in this case, and the *MSGDSC*, *BUFLEN*, *BUFFER* and *DATLEN* parameters are not checked or altered.
3. If the application issuing the MQGET call is running as an IBM MQ MQI client, it is possible for the message retrieved to be lost if during the processing of the MQGET call the IBM MQ MQI client terminates abnormally or the client connection is severed. This arises because the surrogate that is running on the platform of the queue manager and which issues the MQGET call on the behalf of the

client cannot detect the loss of the client until the surrogate is about to return the message to the client; this is after the message has been removed from the queue. This can occur for both persistent messages and nonpersistent messages.

The risk of losing messages in this way can be eliminated by always retrieving messages within units of work (that is, by specifying the GMSYP option on the MQGET call, and using the MQCMIT or MQBACK calls to commit or back out the unit of work when processing of the message is complete). If GMSYP is specified, and the client terminates abnormally or the connection is severed, the surrogate backs out the unit of work on the queue manager and the message is reinstated on the queue.

In principle, the same situation can arise with applications that are running on the platform of the queue manager, but in this case the window during which a message can be lost is small. However, as with IBM MQ MQI clients the risk can be eliminated by retrieving the message within a unit of work.

4. If an application puts a sequence of messages on a particular queue within a single unit of work, and then commits that unit of work successfully, the messages become available for retrieval as follows:
 - If the queue is a *nonshared* queue (that is, a local queue), all messages within the unit of work become available at the same time.
 - If the queue is a *shared* queue, messages within the unit of work become available in the order in which they were put, but not all at the same time. When the system is heavily laden, it is possible for the first message in the unit of work to be retrieved successfully, but for the MQGET call for the second or subsequent message in the unit of work to fail with RC2033. If this occurs, the application must wait a short while and then try the operation again.
5. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if certain conditions are satisfied. See the usage notes in the description of the MQPUT call for details. If the conditions are satisfied, the messages are presented to the receiving application in the order in which they were sent, if:

- Only one receiver is getting messages from the queue.

If there are two or more applications getting messages from the queue, they must agree with the sender the mechanism to be used to identify messages that belong to a sequence. For example, the sender might set all of the *MDCID* fields in the messages in a sequence to a value that was unique to that sequence of messages.

- The receiver does not deliberately change the order of retrieval, for example by specifying a particular *MDMID* or *MDCID*.

If the sending application put the messages as a message group, the messages are presented to the receiving application in the correct order if the receiving application specifies the GMLOGO option on the MQGET call. For more information about message groups, see:

- *MDMFL* field in MQMD
- PMLOGO option in MQPMO
- GMLOGO option in MQGMO

6. Applications test for the feedback code FBQUIT in the *MDFB* field of the *MSGDSC* parameter. If this value is found, the application ends. See the *MDFB* field described in [“MQMD - Message descriptor” on page 1068](#) for more information.
7. If the queue identified by *HOBJ* was opened with the OOSAVA option, and the completion code from the MQGET call is CCOK or CCWARN, the context associated with the queue handle *HOBJ* is set to the context of the message that has been retrieved (unless the GMBRWF or GMBRWN option is set, in which case the context is marked as not available). This context can be used on a subsequent MQPUT or MQPUT1 call by specifying the PMPASI or PMPASA options. This enables the context of the message received to be transferred in whole or in part to another message (for example, when the message is forwarded to another queue). For more information about message context, see [Message context](#) and [Controlling context information](#).

8. If the *GMCONV* option is included in the *GMO* parameter, the application message data is converted to the representation requested by the receiving application, before the data is placed in the *BUFFER* parameter:
 - The *MDFMT* field in the control information in the message identifies the structure of the application data, and the *MDCSI* and *MDENC* fields in the control information in the message specify its character-set identifier and encoding.
 - The application issuing the *MQGET* call specifies in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter the character-set identifier and encoding to which the application message data must be converted.

When conversion of the message data is necessary, the conversion is performed either by the queue manager itself or by a user-written exit, depending on the value of the *MDFMT* field in the control information in the message:

- The following formats are converted automatically by the queue manager; these formats are called "built-in" formats:

FMADMN	FMMDE
FMCICS	FMPCF
FMCMND1	FMRMH
FMCMND2	FMRFH
FMDLH	FMRFH2
FMDH	FMSTR
FMEVNT	FMTM
FMIMS	FMXQH
FMIMVS	

- The format name *FMNONE* is a special value that indicates that the nature of the data in the message is undefined. As a consequence, the queue manager does not attempt conversion when the message is retrieved from the queue.

Note: If *GMCONV* is specified on the *MQGET* call for a message that has a format name of *FMNONE*, and the character set or encoding of the message differs from that specified in the *MSGDSC* parameter, the message is still returned in the *BUFFER* parameter (assuming no other errors), but the call completes with completion code *CCWARN* and reason code *RC2110*.

FMNONE can be used either when the nature of the message data means that it does not require conversion, or when the sending and receiving applications have agreed between themselves the form in which the message data should be sent.

- All other format names cause the message to be passed to a user-written exit for conversion. The exit has the same name as the format, apart from environment-specific additions. User-specified format names must not begin with the letters "MQ", as such names might conflict with format names supported in the future.

User data in the message can be converted between any supported character sets and encodings. However, be aware that if the message contains one or more IBM MQ header structures, the message cannot be converted from or to a character set that has double-byte or multi-byte characters for any of the characters that are valid in queue names. Reason code *RC2111* or *RC2115* results if this is attempted, and the message is returned unconverted. Unicode character set *UCS-2* is an example of such a character set.

On return from *MQGET*, the following reason code indicates that the message was converted successfully:

- *RCNONE*

The following reason code indicates that the message might have been converted successfully; the application must check the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to find out:

- RC2079

All other reason codes indicate that the message was not converted.

Note: The interpretation of the reason code described in this example is true for conversions performed by user-written exits *only* if the exit conforms to the processing guidelines.

9. For the built-in formats listed previously, the queue manager might perform *default conversion* of character strings in the message when the GMCONV option is specified. Default conversion allows the queue manager to use an installation-specified default character set that approximates the actual character set, when converting string data. As a result, the MQGET call can succeed with completion code CCOK, instead of completing with CCWARN and reason code RC2111 or RC2115.

Note: The result of using an approximate character set to convert string data is that some characters might be converted incorrectly. This can be avoided by using in the string only characters which are common to both the actual character set and the default character set.

Default conversion applies both to the application message data and to character fields in the MQMD and MQMDE structures:

- Default conversion of the application message data occurs only when *all* of the following are true:
 - The application specifies GMCONV.
 - The message contains data that must be converted either from or to a character set which is not supported.
 - Default conversion was enabled when the queue manager was installed or restarted.
 - Default conversion of the character fields in the MQMD and MQMDE structures occurs as necessary, if default conversion is enabled for the queue manager. The conversion is performed even if the GMCONV option is not specified by the application on the MQGET call.
10. The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter must be declared instead as a structure, or as a field in a physical file.

Declaring the parameter as a structure increases the maximum length possible to 9999 bytes, while declaring the parameter as a field in a physical file increases the maximum length possible to approximately 32 KB.

Parameters

The MQGET call has the following parameters:

HCONN (10-digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HOBJ (10-digit signed integer) - input

Object handle.

This handle represents the queue from which a message is to be retrieved. The value of *HOBJ* was returned by a previous *MQOPEN* call. The queue must have been opened with one or more of the following options (see [“MQOPEN - Open object”](#) on page 1278 for details):

- *OOINPS*
- *OOINPX*
- *OOINPQ*
- *OOBRW*

MSGDSC (MQMD) - input/output

Message descriptor.

This structure describes the attributes of the message required, and the attributes of the message retrieved. See [“MQMD - Message descriptor”](#) on page 1068 for details.

If *BUFLen* is less than the message length, *MSGDSC* is still entered by the queue manager, whether *GMATM* is specified on the *GMO* parameter (see the *GMOPT* field described in [“MQGMO - Get-message options”](#) on page 1035).

If the application provides a version-1 *MQMD*, the message returned has an *MQMDE* prefixed to the application message data, but *only* if one or more of the fields in the *MQMDE* has a nondefault value. If all of the fields in the *MQMDE* have default values, the *MQMDE* is omitted. A format name of *FMMDE* in the *MDFMT* field in *MQMD* indicates that an *MQMDE* is present.

GMO (MQGMO) - input/output

Options that control the action of *MQGET*.

See [“MQGMO - Get-message options”](#) on page 1035 for details.

BUFLen (10-digit signed integer) - input

Length in bytes of the *BUFFER* area.

Zero can be specified for messages that have no data, or if the message is to be removed from the queue and the data discarded (*GMATM* must be specified in this case).

Note: The length of the longest message that it is possible to read from the queue is given by the *MaxMsgLength* queue attribute; see [“Attributes for queues”](#) on page 1324.

BUFFER (1-byte bit string x BUFLen) - output

Area to contain the message data.

The buffer must be aligned on a boundary appropriate to the nature of the data in the message. 4-byte alignment must be suitable for most messages (including messages containing IBM MQ header structures), but some messages might require more stringent alignment. For example, a message containing a 64-bit binary integer might require 8-byte alignment.

If *BUFLen* is less than the message length, as much of the message as possible is moved into *BUFFER* ; this happens whether *GMATM* is specified on the *GMO* parameter (see the *GMOPT* field described in [“MQGMO - Get-message options”](#) on page 1035 for more information).

The character set and encoding of the data in *BUFFER* are given by the *MDCSI* and *MDENC* fields returned in the *MSGDSC* parameter. If these values are different from the values required by the receiver, the receiver must convert the application message data to the character set and encoding required. The *GMCONV* option can be used with a user-written exit to perform the conversion of the message data (see [“MQGMO - Get-message options”](#) on page 1035 for details of this option).

Note: All of the other parameters on the *MQGET* call are in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue manager attribute and *ENNAT*).

If the call fails, the contents of the buffer might still have changed.

DATLEN (10-digit signed integer) - output

Length of the message.

This is the length in bytes of the application data *in the message*. If this message length is greater than *BUFLen*, only *BUFLen* bytes are returned in the *BUFFER* parameter (that is, the message is truncated). If the value is zero, it means that the message contains no application data.

If *BUFLen* is less than the message length, *DATLEN* is still entered by the queue manager, whether *GMATM* is specified on the *GMO* parameter (see the *GMOPT* field described in “[MQGMO - Get-message options](#)” on page 1035 for more information). This allows the application to determine the size of the buffer required to accommodate the message data, and then reissue the call with a buffer of the appropriate size.

However, if the *GMCONV* option is specified, and the converted message data is too long to fit in *BUFFER*, the value returned for *DATLEN* is:

- The length of the *unconverted* data, for queue manager defined formats.

In this case, if the nature of the data causes it to expand during conversion, the application must allocate a buffer bigger than the value returned by the queue manager for *DATLEN*.

- The value returned by the data-conversion exit, for application-defined formats.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

The following reason codes are the ones that the queue manager can return for the *REASON* parameter. If the application specifies the *GMCONV* option, and a user-written exit is invoked to convert some or all of the message data, it is the exit that decides what value is returned for the *REASON* parameter. As a result, values other than the values documented later in this section are possible.

If *CMPCOD* is *CCOK*:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is *CCWARN*:

RC2120

(2120, X'848') Converted data too large for buffer.

RC2190

(2190, X'88E') Converted string too large for field.

RC2150

(2150, X'866') DBCS string not valid.

RC2110

(2110, X'83E') Message format not valid.

RC2243

(2243, X'8C3') Message segments have differing CCSIDs.

RC2244

(2244, X'8C4') Message segments have differing encodings.

RC2209

(2209, X'8A1') No message locked.

RC2119

(2119, X'847') Message data not converted.

RC2272

(2272, X'8E0') Message data partially converted.

RC2145

(2145, X'861') Source buffer parameter not valid.

RC2111

(2111, X'83F') Source coded character set identifier not valid.

RC2113

(2113, X'841') Packed-decimal encoding in message not recognized.

RC2114

(2114, X'842') Floating-point encoding in message not recognized.

RC2112

(2112, X'840') Source integer encoding not recognized.

RC2143

(2143, X'85F') Source length parameter not valid.

RC2146

(2146, X'862') Target buffer parameter not valid.

RC2115

(2115, X'843') Target coded character set identifier not valid.

RC2117

(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

RC2118

(2118, X'846') Floating-point encoding specified by receiver not recognized.

RC2116

(2116, X'844') Target integer encoding not recognized.

RC2079

(2079, X'81F') Truncated message returned (processing completed).

RC2080

(2080, X'820') Truncated message returned (processing not completed).

If *CMPCOD* is CCFAIL:

RC2004

(2004, X'7D4') Buffer parameter not valid.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2010

(2010, X'7DA') Data length parameter not valid.

RC2016

(2016, X'7E0') Gets inhibited for the queue.

RC2186

(2186, X'88A') Get-message options structure not valid.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019
(2019, X'7E3') Object handle not valid.

RC2241
(2241, X'8C1') Message group not complete.

RC2242
(2242, X'8C2') Logical message not complete.

RC2259
(2259, X'8D3') Inconsistent browse specification.

RC2245
(2245, X'8C5') Inconsistent unit-of-work specification.

RC2246
(2246, X'8C6') Message under cursor not valid for retrieval.

RC2247
(2247, X'8C7') Match options not valid.

RC2026
(2026, X'7EA') Message descriptor not valid.

RC2250
(2250, X'8CA') Message sequence number not valid.

RC2033
(2033, X'7F1') No message available.

RC2034
(2034, X'7F2') Browse cursor not positioned on message.

RC2036
(2036, X'7F4') Queue not open for browse.

RC2037
(2037, X'7F5') Queue not open for input.

RC2041
(2041, X'7F9') Object definition changed since opened.

RC2101
(2101, X'835') Object damaged.

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2052
(2052, X'804') Queue has been deleted.

RC2058
(2058, X'80A') Queue manager name not valid or not known.

RC2059
(2059, X'80B') Queue manager not available for connection.

RC2161
(2161, X'871') Queue manager quiescing.

RC2162
(2162, X'872') Queue manager shutting down.

RC2102
(2102, X'836') Insufficient system resources available.

RC2071
(2071, X'817') Insufficient storage available.

RC2024
(2024, X'7E8') No more messages can be handled within current unit of work.

RC2072
(2072, X'818') Syncpoint support not available.

RC2195

(2195, X'893') Unexpected error occurred.

RC2255

(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2090

(2090, X'82A') Wait interval in MQGMO not valid.

RC2256

(2256, X'8D0') Wrong version of MQGMO supplied.

RC2257

(2257, X'8D1') Wrong version of MQMD supplied.

RPG Declaration

```

C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQGET(HCONN : HOBJ : MSGDSC : GMO :
C                      BUFLN : BUFFER : DATLEN :
C                      CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQGET          PR          EXTPROC('MQGET')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object handle
D HOBJ          10I 0 VALUE
D* Message descriptor
D MSGDSC          364A
D* Options that control the action of MQGET
D GMO          112A
D* Length in bytes of the Buffer area
D BUFLN          10I 0 VALUE
D* Area to contain the message data
D BUFFER          * VALUE
D* Length of the message
D DATLEN          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0

```

MQINQ - Inquire about object attributes

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object.

The following types of object are valid:

- Queue
- Namelist
- Process definition
- Queue manager
- [“Syntax” on page 1261](#)
- [“Usage notes” on page 1262](#)
- [“Parameters” on page 1263](#)
- [“RPG Declaration” on page 1270](#)

Syntax

MQINQ (*HCONN*, *HOBJ*, *SELCNT*, *SELS*, *IACNT*, *INTATR*, *CALEN*, *CHRATR*, *CMPCOD*, *REASON*)

Usage notes

1. The values returned are a snapshot of the selected attributes. There is no guarantee that the attributes are not changed before the application can act upon the returned values.
2. When you open a model queue, a dynamic local queue is created. This is true even if you open the model queue to inquire about its attributes.

The attributes of the dynamic queue (with certain exceptions) are the same as those of the model queue at the time the dynamic queue is created. If you then use the MQINQ call on this queue, the queue manager returns the attributes of the dynamic queue, and not those of the model queue. See [Table 1](#) for details of which attributes of the model queue are inherited by the dynamic queue.

3. If the object being inquired is an alias queue, the attribute values returned by the MQINQ call are those of the alias queue, and not those of the base queue to which the alias resolves.
4. If the object being inquired is a cluster queue, the attributes that can be inquired depend on how the queue is opened:
 - If the cluster queue is opened for inquire plus one or more of input, browse, or set, there must be a local instance of the cluster queue in order for the open to succeed. In this case the attributes that can be inquired are those valid for local queues.
 - If the cluster queue is opened for inquire alone, or inquire and output, only the following attributes can be inquired; the *QType* attribute has the value QTCLUS in this case:
 - CAQD
 - CAQN
 - IADBND
 - IADPER
 - IADPRI
 - IAIPUT
 - IAQTYP

If the cluster queue is opened with no fixed binding (that is, OOBNDN specified on the MQOPEN call, or OOBNDQ specified when the *DefBind* attribute has the value BNDNOT), successive MQINQ calls for the queue might inquire different instances of the cluster queue, although typically all of the instances have the same attribute values.

For more information about cluster queues, see [Configuring a queue manager cluster](#).

5. If a number of attributes are to be inquired, and then some of them are to be set using the MQSET call, it might be convenient to position at the beginning of the selector arrays the attributes that are to be set, so that the same arrays (with reduced counts) can be used for MQSET.
6. If more than one of the warning situations arise (see the *CMPCOD* parameter), the reason code returned is the *first* one in the following list that applies:
 - a. RC2068
 - b. RC2022
 - c. RC2008
7. For more information about object attributes, see:
 - [“Attributes for queues” on page 1324](#)
 - [“Attributes for namelists” on page 1352](#)
 - [“Attributes for process definitions” on page 1353](#)
 - [“Attributes for the queue manager” on page 1355](#)
8. A new local queue SYSTEM.ADMIN.COMMAND.EVENT is used for queuing messages that are generated whenever commands are issued. Messages are put onto this queue for most commands, depending on how the CMDEV queue manager attribute is set:

- **ENABLED** - command event messages are generated and put onto the queue for all successful commands.
- **NODISPLAY** - command event messages are generated and put onto the queue for all successful commands other than the **DISPLAY (MQSC)** command, and the **Inquire (PCF)** command.
- **DISABLED** - command event messages are not generated (this is the queue manager's initial default value).

Parameters

The **MQINQ** call has the following parameters:

HCONN (10 digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous **MQCONN** or **MQCONNEX** call.

On IBM i for applications running in compatibility mode, the **MQCONN** call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HOBJ (10 digit signed integer) - input

Object handle.

This handle represents the object (of any type) with attributes that are required. The handle must have been returned by a previous **MQOPEN** call that specified the **OOINQ** option.

SELCNT (10 digit signed integer) - input

Count of selectors.

This is the count of selectors that are supplied in the *SELS* array. It is the number of attributes that are to be returned. Zero is a valid value. The maximum number allowed is 256.

SELS (10 digit signed integer x SELCNT) - input

Array of attribute selectors.

This is an array of *SELCNT* attribute selectors; each selector identifies an attribute (integer or character) with a value that is required.

Each selector must be valid for the type of object that *HOBJ* represents, otherwise the call fails with completion code **CCFAIL** and reason code **RC2067**.

In the special case of queues:

- If the selector is not valid for queues of *any* type, the call fails with completion code **CCFAIL** and reason code **RC2067**.
- If the selector is applicable *only* to queues of type or types other than that of the object, the call succeeds with completion code **CCWARN** and reason code **RC2068**.
- If the queue being inquired is a cluster queue, the selectors that are valid depend on how the queue was resolved; see usage note 4 for further details.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (**IA*** selectors) are returned in *INTATR* in the same order in which these selectors occur in *SELS*. Attribute values that correspond to character attribute selectors (**CA*** selectors) are returned in

CHRATR in the same order in which those selectors occur. IA* selectors can be interleaved with the CA* selectors; only the relative order within each type is important.

Note:

1. The integer and character attribute selectors are allocated within two different ranges; the IA* selectors reside within the range IAFRST through IALAST, and the CA* selectors within the range CAFRST through CALAST.

For each range, the constants IALSTU and CALSTU define the highest value that the queue manager accepts.

2. If all the IA* selectors occur first, the same element numbers can be used to address corresponding elements in the *SELS* and *INTATR* arrays.

The attributes that can be inquired are listed in the following tables. For the CA* selectors, the constant that defines the length in bytes of the resulting string in *CHRATR* is given in parentheses.

Table 621. MQINQ attribute selectors for queues.		
See the bottom of the table for an explanation of the notes.		
Selector	Description	Note
CAALTD	Date of most recent alteration (LNDATE).	1
CAALTT	Time of most recent alteration (LNTIME).	1
CABRQN	Excessive backout-requeue name (LNQN).	5
CABASQ	Name of queue that alias resolves to (LNQN).	
CACFSN	Coupling-facility structure name (LNCFSN).	3
CACLN	Cluster name (LNCLUN).	1
CACLNL	Cluster namelist (LNNLN).	1
CACRTD	Queue creation date (LNCRTD).	
CACRTT	Queue creation time (LNCRTT).	
CAINIQ	Initiation queue name (LNQN).	
CAPRON	Name of process definition (LNPRON).	
CAQD	Queue description (LNQD).	
CAQN	Queue name (LNQN).	
CARQMN	Name of remote queue manager (LNQMN).	
CARQN	Name of remote queue as known on remote queue manager (LNQN).	
CATRGD	Trigger data (LNTRGD).	5
CAXQN	Transmission queue name (LNQN).	
IABTHR	Backout threshold.	5
IACDEP	Number of messages on queue.	
IADBND	Default binding.	1
IADINP	Default open-for-input option.	5
IADPER	Default message persistence.	
IADPRI	Default message priority.	5
IADEFT	Queue definition type.	

Table 621. MQINQ attribute selectors for queues.

See the bottom of the table for an explanation of the notes.

(continued)

Selector	Description	Note
IADIST	Distribution list support.	2
IAHGB	Whether to harden backout count.	5
IAIGET	Whether get operations are allowed.	
IAIPUT	Whether put operations are allowed.	
IAMLEN	Maximum message length.	
IAMDEP	Maximum number of messages allowed on queue.	
IAMDS	Whether message priority is relevant.	5
IAOIC	Number of MQOPEN calls that have the queue open for input.	
IAOOC	Number of MQOPEN calls that have the queue open for output.	
IAQDHE	Control attribute for queue depth high events.	4, 5
IAQDHL	High limit for queue depth.	4, 5
IAQDLE	Control attribute for queue depth low events.	4, 5
IAQDLL	Low limit for queue depth.	4, 5
IAQDME	Control attribute for queue depth max events.	4, 5
IAQSI	Limit for queue service interval.	4, 5
IAQSIE	Control attribute for queue service interval events.	4, 5
IAQTYP	Queue type.	
IAQSGD	Queue-sharing group disposition.	3
IARINT	Queue retention interval.	5
IASCOP	Queue definition scope.	4, 5
IASHAR	Whether queue can be shared for input.	
IATRGC	Trigger control.	
IATRGD	Trigger depth.	5
IATRGP	Threshold message priority for triggers.	5
IATRGT	Trigger type.	
IAUSAG	Usage.	
CLWLUSEQ	Use remote queues.	

Note:

1. Supported on AIX, HP-UX, z/OS, IBM i, Solaris, Windows, plus IBM MQ MQI clients connected to these systems.
2. Supported on AIX, HP-UX, IBM i, Solaris, Windows, plus IBM MQ clients connected to these systems.
3. Supported on z/OS.

4. Not supported on z/OS.

5. Not supported on VSE/ESA.

Table 622. MQINQ attribute selectors for namelists.

See [notes](#) for an explanation of the notes.

Selector	Description	Note
CAALTD	Date of most recent alteration (LNDATE)	1
CAALTT	Time of most recent alteration (LNTIME)	1
CALSTD	Namelist description (LNNLD)	1
CALSTN	Name of namelist object (LNNLN)	1
CANAMS	Names in the namelist (LNQN x Number of names in the list)	1
IANAMC	Number of names in the namelist	1
IAQSGD	Queue-sharing group disposition	3

Table 623. MQINQ attribute selectors for process definitions.

See [notes](#) for an explanation of the notes.

Selector	Description	Note
CAALTD	Date of most recent alteration (LNDATE)	1
CAALTT	Time of most recent alteration (LNTIME)	1
CAAPPI	Application identifier (LNPROA)	5
CAENV D	Environment data (LNPROE)	5
CAPROD	Description of process definition (LNPROD)	5
CAPRON	Name of process definition (LNPRON)	5
CAUSR D	User data (LNPROU)	5
IAAPPT	Application type	5
IAQSGD	Queue-sharing group disposition	3

Table 624. MQINQ attribute selectors for the queue manager.

See [notes](#) for an explanation of the notes.

Selector	Description	Note
CAALTD	Date of most recent alteration (LNDATE)	1
CAALTT	Time of most recent alteration (LNTIME)	1
CACADX	Automatic channel definition exit name (LNEXN)	1
CACLWD	Data passed to cluster workload exit (LNEXDA)	1
CACLWX	Name of cluster workload exit (LNEXN)	1
CACMDQ	System command input queue name (LNQN)	5
CADLQ	Name of dead-letter queue (LNQN)	5
CADXQN	Default transmission queue name (LNQN)	5

Table 624. MQINQ attribute selectors for the queue manager.

See [notes](#) for an explanation of the notes.

(continued)

Selector	Description	Note
CAQMD	Queue manager description (LNQMD)	5
CAQMID	Queue-manager identifier (LNQMID)	1
CAQMN	Name of local queue manager (LNQMN)	5
CAQSGN	Queue-sharing group name (LNQSGN)	3
CARPN	Name of cluster for which queue manager provides repository services (LNQMN)	1
CARPNL	Name of namelist object containing names of clusters for which queue manager provides repository services (LNNLN)	1
CMDEV	Control attribute that determines whether messages generated when commands are issued, are put onto a queue	8
IAAUTE	Control attribute for authority events	4, 5
IACAD	Control attribute for automatic channel definition	2
IACADE	Control attribute for automatic channel definition events	2
IACLXQ	Default cluster transmission queue type	4
IACLWL	Cluster workload length	1
IACCSI	Coded character set identifier	5
IACMDL	Command level supported by queue manager	5
IACFGE	Control attribute for configuration events	3
IADIST	Distribution list support	2
IAINHE	Control attribute for inhibit events	4, 5
IALCLE	Control attribute for local events	4, 5
IAMHND	Maximum number of handles	5
IAMLEN	Maximum message length	5
IAMPRI	Maximum priority	5
IAMUNC	Maximum number of uncommitted messages within a unit of work	5
IAPFME	Control attribute for performance events	4, 5
IAPLAT	Platform on which the queue manager resides	5
IARMTE	Control attribute for remote events	4, 5
IASSE	Control attribute for start stop events	4, 5
IASYNC	Sync point availability	5
IATRLFT	Lifetime of unused non-administrative topics	
IATRGI	Trigger interval	5

IACNT (10 digit signed integer) - input

Count of integer attributes.

This is the number of elements in the *INTATR* array. Zero is a valid value.

If this is at least the number of IA* selectors in the *SELS* parameter, all integer attributes requested are returned.

INTATR (10 digit signed integer x IACNT) - output

Array of integer attributes.

This is an array of *IACNT* integer attribute values.

Integer attribute values are returned in the same order as the IA* selectors in the *SELS* parameter. If the array contains more elements than the number of IA* selectors, the excess elements are unchanged.

If *HOBj* represents a queue, but an attribute selector is not applicable to that type of queue, the specific value *IAVNA* is returned for the corresponding element in the *INTATR* array.

CALEN (10 digit signed integer) - input

Length of character attributes buffer.

This is the length in bytes of the *CHRATR* parameter.

This must be at least the sum of the lengths of the requested character attributes (see *SELS*). Zero is a valid value.

CHRATR (1 byte character string x CALEN) - output

Character attributes.

This is the buffer in which the character attributes are returned, concatenated together. The length of the buffer is given by the *CALEN* parameter.

Character attributes are returned in the same order as the CA* selectors in the *SELS* parameter. The length of each attribute string is fixed for each attribute (see *SELS*), and the value in it is padded to the right with blanks if necessary. If the buffer is larger than that needed to contain all of the requested character attributes (including padding), the bytes beyond the last attribute value returned are unchanged.

If *HOBj* represents a queue, but an attribute selector is not applicable to that type of queue, a character string consisting entirely of asterisks (*) is returned as the value of that attribute in *CHRATR*.

CMPCOD (10 digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10 digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2008

(2008, X'7D8') Not enough space allowed for character attributes.

RC2022

(2022, X'7E6') Not enough space allowed for integer attributes.

RC2068

(2068, X'814') Selector not applicable to queue type.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2006

(2006, X'7D6') Length of character attributes not valid.

RC2007

(2007, X'7D7') Character attributes string not valid.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2021

(2021, X'7E5') Count of integer attributes not valid.

RC2023

(2023, X'7E7') Integer attributes array not valid.

RC2038

(2038, X'7F6') Queue not open for inquire.

RC2041

(2041, X'7F9') Object definition changed since opened.

RC2101

(2101, X'835') Object damaged.

RC2052

(2052, X'804') Queue has been deleted.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2065

(2065, X'811') Count of selectors not valid.

RC2067

(2067, X'813') Attribute selector not valid.

RC2066

(2066, X'812') Count of selectors too large.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```
D*.1.....2.....3.....4.....5.....6.....7..  
C                                CALLP      MQINQ(HCONN : HOBJ : SELCNT :  
C                                SELS(1) : IACNT : INTATR(1) :  
C                                CALEN : CHRATR : CMPCOD :  
C                                REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..  
DMQINQ                                PR      EXTPROC('MQINQ')  
D* Connection handle  
D HCONN                                10I 0 VALUE  
D* Object handle  
D HOBJ                                10I 0 VALUE  
D* Count of selectors  
D SELCNT                              10I 0 VALUE  
D* Array of attribute selectors  
D SELS                                10I 0  
D* Count of integer attributes  
D IACNT                                10I 0 VALUE  
D* Array of integer attributes  
D INTATR                              10I 0  
D* Length of character attributes buffer  
D CALEN                                10I 0 VALUE  
D* Character attributes  
D CHRATR                                *   VALUE  
D* Completion code  
D CMPCOD                              10I 0  
D* Reason code qualifying CMPCOD  
D REASON                              10I 0
```

MQINQMP - Inquire message property

The MQINQMP call returns the value of a property of a message.

- [“Syntax” on page 1270](#)
- [“Parameters” on page 1270](#)
- [“RPG Declaration” on page 1274](#)

Syntax

MQINQMP (*Hconn*, *Hmsg*, *InqPropOpts*, *Name*, *PropDesc*, *Type*, *ValueLength*, *Value*, *DataLength*, *CompCode*, *Reason*)

Parameters

The MQINQMP call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *Hconn* must match the connection handle that was used to create the message handle specified in the *Hmsg* parameter.

If the message handle was created using HCUNAS then a valid connection must be established on the thread inquiring a property of the message handle, otherwise the call fails with RC2009.

HMSG (20-digit signed integer) - input

This is the message handle to be inquired. The value was returned by a previous **MQCRTMH** call.

INQOPT (MQIMPO) - input

See the [MQIMPO](#) data type for details.

PRNAME (MQCHARV) - input

This describes the name of the property to inquire.

If no property with this name can be found, the call fails with reason RC2471.

You can use the percent sign (%) character at the end of the property name. The wildcard matches zero or more characters, including the period (.) character. This allows an application to inquire the value of many properties. Call MQINQMP with option IPINQF to get the first matching property and again with the option IPINQN to get the next matching property. When no more matching properties are available, the call fails with RC2471. If the *ReturnedName* field of the InqPropOpts structure is initialized with an address or offset for the returned name of the property, this is completed on return from MQINQMP with the name of the property that has been matched. If the *VSBuFSIZE* field of the *ReturnedName* in the InqPropOpts structure is less than the length of the returned property name the completion code is set CCFAIL with reason RC2465.

Properties that have known synonyms are returned as follows:

1. Properties with the prefix "mqps." are returned with the IBM MQ property name. For example, "MQTopicString" is the returned name rather than "mqps.Top".
2. Properties with the prefix "jms." or "mcd." are returned as the JMS header field name. For example, "JMSExpiration" is the returned name rather than "jms.Exp".
3. Properties with the prefix "usr." are returned without that prefix. For example, "Color" is returned rather than "usr.Color".

Properties with synonyms are only returned once.

In the RPG programming language, the following macro variables are defined for inquiring on all properties and all properties that begin "usr.":

INQALL

Inquire on all properties of the message.

INQUSR

Inquire on all properties of the message that start "usr.". The returned name is returned without the "usr." prefix.

If IPINQN is specified but Name has changed since the previous call or this is the first call, then IPINQF is implied.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

PRPDSC (MQPD) - output

This structure is used to define the attributes of a property, including what happens if the property is not supported, what message context the property belongs to, and what messages the property should be copied into. See [MQPD](#) for details of this structure.

TYPE (10-digit signed integer) - input/output

On return from the MQINQMP call this parameter is set to the data type of *Value*. The data type can be any of the following:

TYPBOL

A boolean.

TYPBST

a byte string.

TYPI8

An 8-bit signed integer.

TYPI16

A 16-bit signed integer.

TYPI32

A 32-bit signed integer.

TYPI64

A 64-bit signed integer.

TYPF32

A 32-bit floating-point number.

TYPF64

A 64-bit floating-point number.

TYPSTR

A character string.

TYPNUL

The property exists but has a null value.

If the data type of the property value is not recognized then TYPSTR is returned and a string representation of the value is placed into the *Value* area. A string representation of the data type can be found in the *IPCTYP* field of the *IPOPT* parameter. A warning completion code is returned with reason RC2467.

Additionally, if the option IPCTYP is specified, conversion of the property value is requested. Use *Type* as an input to specify the data type that you want the property to be returned as. See the description of the IPCTYP option of the [“MQIMPO - Inquire message property options”](#) on page 1061 for details of data type conversion.

If you do not request type conversion, you can use the following value on input:

TYPAST

The value of the property is returned without converting its data type.

VALLEN (10-digit signed integer) - input

The length in bytes of the Value area.

Specify zero for properties that you do not require the value returned for. These could be properties which are designed by an application to have a null value or an empty string. Also specify zero if the IPQLEN option has been specified; in this case no value is returned.

VALUE (1-byte bit string x VALLEN) - output

This is the area to contain the inquired property value. The buffer should be aligned on a boundary appropriate for the value being returned. Failure to do so might result in an error when the value is later accessed.

If *VALLEN* is less than the length of the property value, as much of the property value as possible is moved into *VALUE* and the call fails with completion code CCFAIL and reason RC2469.

The character set of the data in *VALUE* is given by the IPRETCSI field in the INQOPT parameter. The encoding of the data in *VALUE* is given by the IPRETENC field in the INQOPT parameter.

If the *VALLEN* parameter is zero, *VALUE* is not referred to.

DATLEN (10-digit signed integer) - output

This is the length in bytes of the actual property value as returned in the *Value* area.

If *DataLength* is less than the property value length, *DataLength* is still entered on return from the MQINQMP call. This allows the application to determine the size of the buffer required to accommodate the property value, and then reissue the call with a buffer of the appropriate size.

The following values may also be returned.

If the *Type* parameter is set to TYPSTR or TYPBST:

VLEMP

The property exists but contains no characters or bytes.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CompCode*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CompCode* is CCWARN:

RC2492

(2492, X'09BC') Returned property name not converted.

RC2466

(2466, X'09A2') Property value not converted.

RC2467

(2467, X'09A3') Property data type is not supported.

RC2421

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'0852') Unable to load adapter service module.

RC2157

(2157, X'086D') Primary and home ASIDs differ.

RC2004

(2004, X'07D4') Value parameter not valid.

RC2005

(2005, X'07D5') Value length parameter not valid.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2009

(2009, X'07D9') Connection to queue manager lost.

RC2010

(2010, X'07DA') Data length parameter not valid.

RC2464

(2464, X'09A0') Inquire message property options structure not valid.

RC2460

(2460, X'099C') Message handle not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2064

(2046, X'07F8') Options not valid or not consistent.

RC2482

(2482, X'09B2') Property descriptor structure not valid.

RC2470

(2470, X'09A6') Conversion from the actual to requested data type not supported.

RC2442

(2442, X'098A') Invalid property name.

RC2465

(2465, X'09A1') Property name too large for returned name buffer.

RC2471

(2471, X'09A7') Property not available.

RC2469

(2469, X'09A5') Property value too large for the Value area.

RC2472

(2472, X'09A8') Number format error encountered in value data.

RC2473

(2473, X'09A9') Invalid requested property type.

RC2111

(2111, X'083F') Property name coded character set identifier not valid.

RC2071

(2071, X'0871') Insufficient storage available.

RC2195

(2195, X'0893') Unexpected error occurred.

For detailed information about these codes, see:

- [IBM MQ for z/OS messages, completion, and reason codes for IBM MQ for z/OS](#)
- [Messages and reason codes for all other IBM MQ platforms](#)

RPG Declaration

```

C*..1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQINQMP(HCONN : HMSG : INQOPT :
                                           PRNAME : PRPDSC : TYPE :
                                           VALLEN : VALUE : DATLEN :
                                           CMPCOD : REASON)

```

The prototype definition for the call is:

```

DMQINQMP          PR          EXTPROC('MQINQMP')
D* Connection handle
D HCONN            10I 0 VALUE
D* Message handle
D HMSG             20I 0 VALUE
D* Options that control the action of MQINQMP
D INQOPT           72A
D* Property name
D PRNAME           32A
D* Property descriptor
D PRPDSC           24A
D* Property data type
D TYPE            10I 0
D* Length in bytes of the Value area
D VALLEN           10I 0 VALUE
D* Property value
D VALUE            *   VALUE
D* Length of the property value
D DATLEN           10I 0
D* Completion code
D CMPCOD           10I 0
D* Reason code qualifying CompCode
D REASON           10I 0

```

MQMHBUF - Convert message handle into buffer

The MQMHBUF converts a message handle into a buffer and is the inverse of the MQBUFMH call.

- [“Syntax” on page 1275](#)

- [“Usage notes” on page 1275](#)
- [“Parameters” on page 1275](#)
- [“RPG Declaration” on page 1277](#)

Syntax

MQMHBUF (*Hconn, Hmsg, MsgHBufOpts, Name, MsgDesc, BufferLength, Buffer, DataLength, CompCode, Reason*)

Usage notes

MQMHBUF converts a message handle into a buffer.

You can use it with an MQGET API exit to access certain properties, by using the message property APIs, and then pass these properties in a buffer back to an application designed to use MQRFH2 headers rather than message handles.

This call is the inverse of the MQBUFMH call, which you can use to parse message properties from a buffer into a message handle.

Parameters

The MQMHBUF call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager.

The value of *HCONN* must match the connection handle that was used to create the message handle specified in the *HMSG* parameter.

If the message handle was created by using HCUNAS, a valid connection must be established on the thread deleting the message handle. If a valid connection is not established, the call fails with RC2009.

HMSG (20-digit signed integer) - input

This handle is the message handle for which a buffer is required.

The value was returned by a previous MQCRTMH call.

MHBOPT (MQMHBO) - input

The MQMHBO structure allows applications to specify options that control how buffers are produced from message handles.

See [“MQBMHO - Buffer to message handle options” on page 979](#) for details.

PRNAME (MQCHARV) - input

The name of the property or properties to put into the buffer.

If no property matching the name can be found, the call fails with RC2471.

Wildcards

You can use a wildcard to put more than one property into the buffer. To do so, use the percent sign (%) at the end of the property name. This wildcard matches zero or more characters, including the period (.) character.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

MSGDSC (MQMD) - input/output

The *MSGDSC* structure describes the contents of the buffer area.

On output, the *Encoding*, *CodedCharSetId* and *Format* fields are set to correctly describe the encoding, character set identifier, and format of the data in the buffer area as written by the call.

Data in this structure is in the character set and encoding of the application.

BUFLEN (10-digit signed integer) - input

BUFLEN is the length of the Buffer area, in bytes.

BUFFER (1-byte bit string x BUFLEN) - input/output

BUFFER defines the area containing the message buffer. For most data, you must align the buffer on a 4-byte boundary.

If *BUFFER* contains character or numeric data, set the *CodedCharSetId* and *Encoding* fields in the *MSGDSC* parameter to the values appropriate to the data; this enables the data to be converted, if necessary.

If properties are found in the message buffer they are optionally removed; they later become available from the message handle on return from the call.

In the C programming language, the parameter is declared as a pointer-to-void, which means the address of any type of data can be specified as the parameter.

If the *BUFLEN* parameter is zero, *BUFFER* is not referred to. In this case, the parameter address passed by programs written in C or System/390 assembler can be null.

DATLEN (10-digit signed integer) - output

DATLEN is the length, in bytes, of the returned properties in the buffer. If the value is zero, no properties matched the value given in *PRNAME* and the call fails with reason code RC2471.

If *BUFLEN* is less than the length required to store the properties in the buffer, the MQMHBUF call fails with RC2469, but a value is still entered into *DATLEN*. This allows the application to determine the size of the buffer required to accommodate the properties, and then reissue the call with the required *BUFLEN*.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2501

(2501, X'095C') Message handle to buffer options structure not valid.

RC2004

(2004, X'07D4') Buffer parameter not valid.

RC2005

(2005, X'07D5') Buffer length parameter not valid.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2009

(2009, X'07D9') Connection to queue manager lost.

RC2010

(2010, X'07DA') Data length parameter not valid.

RC2460

(2460, X'099C') Message handle not valid.

RC2026

(2026, X'07EA') Message descriptor not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2442

(2442, X'098A') Property name is not valid.

RC2471

(2471, X'09A7') Property not available.

RC2469

(2469, X'09A5') BufferLength value is too small to contain specified properties.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQMHBUFF(HCONN : HMSG : MHBOPT :
                                                PRNAME : MSGDSC : BUFLen :
                                                BUFFER : DATLEN :
                                                CMPCOD : REASON)

```

The prototype definition for the call is:

```

DMQMHBUFF          PR                EXTPROC('MQMHBUFF')
D* Connection handle
D HCONN              10I 0 VALUE
D* Message handle
D HMSG              20I 0 VALUE
D* Options that control the action of MQMHBUFF
D MHBOPT            12A
D* Property name
D PRNAME            32A
D* Message descriptor
D MSGDSC            364A
D* Length in bytes of the Buffer area
D BUFLen            10I 0 VALUE
D* Area to contain the properties
D BUFFER            *   VALUE
D* Length of the properties
D DATLEN            10I 0
D* Completion code
D CMPCOD            10I 0
D* Reason code qualifying CompCode
D REASON            10I 0

```

MQOPEN - Open object

The MQOPEN call establishes access to an object.

The following types of object are valid:

- Queue (including distribution lists)
- Namelist
- Process definition
- Queue manager
- Topic
- [“Syntax” on page 1278](#)
- [“Usage notes” on page 1278](#)
- [“Parameters” on page 1282](#)
- [“RPG Declaration” on page 1288](#)

Syntax

MQOPEN (*HCONN*, *OBJDSC*, *OPTS*, *HOBJ*, *CMPCOD*, *REASON*)

Usage notes

1. The object opened is one of the following:

- A queue, in order to:
 - Get or browse messages (using the MQGET call)
 - Put messages (using the MQPUT call)
 - Inquire about the attributes of the queue (using the MQINQ call)
 - Set the attributes of the queue (using the MQSET call)

If the queue named is a model queue, a dynamic local queue is created.

A distribution list is a special type of queue object that contains a list of queues. It can be opened to put messages, but not to get or browse messages, or to inquire or set attributes. See usage note 8 for further details.

A queue that has QSGDISP(GROUP) is a special type of queue definition that cannot be used with the MQOPEN or MQPUT1 calls.

- A namelist, in order to:
 - Inquire about the names of the queues in the list (using the MQINQ call).
- A process definition, in order to:
 - Inquire about the process attributes (using the MQINQ call).
- The queue manager, in order to:
 - Inquire about the attributes of the local queue manager (using the MQINQ call).

2. It is valid for an application to open the same object more than once. A different object handle is returned for each open. Each handle that is returned can be used for the functions for which the corresponding open was performed.

3. If the object being opened is a queue but not a cluster queue, all name resolution within the local queue manager takes place at the time of the MQOPEN call. This might include one or more of the following for a particular MQOPEN call:

- Alias resolution to the name of a base queue
- Resolution of the name of a local definition of a remote queue to the name of the remote queue manager, and the name by which the queue is known at the remote queue manager

- Resolution of the remote queue manager name to the name of a local transmission queue

However, be aware that subsequent MQINQ or MQSET calls for the handle relate solely to the name that has been opened, and not to the object resulting after name resolution has occurred. For example, if the object opened is an alias, the attributes returned by the MQINQ call are the attributes of the alias, not the attributes of the base queue to which the alias resolves. Name resolution checking is still carried out, however, regardless of what is specified for the *OPTS* parameter on the corresponding MQOPEN.

If the object being opened is a cluster queue, name resolution can occur at the time of the MQOPEN call, or be deferred until later. The point at which resolution occurs is controlled by the OOBND* options specified on the MQOPEN call:

- OOBND0
- OOBNDN
- OOBNDQ

See [Name resolution](#) for more information about name resolution for cluster queues.

4. The attributes of an object can change while an application has the object open. In many cases, the application does not notice this, but for certain attributes the queue manager marks the handle as no longer valid. These are:

- Any attribute that affects the name resolution of the object. This applies regardless of the open options used, and includes the following:
 - A change to the *BaseQName* attribute of an alias queue that is open.
 - A change to the *RemoteQName* or *RemoteQMGrName* queue attributes, for any handle that is open for this queue, or for a queue which resolves through this definition as a queue manager alias.
 - Any change that causes a currently open handle for a remote queue to resolve to a different *transmission* queue, or to fail to resolve to one at all. For example, this can include:
 - A change to the *XmitQName* attribute of the local definition of a remote queue, whether the definition is being used for a queue, or for a queue manager alias.

There is one exception to this, namely the creation of a new transmission queue. A handle that would have resolved to this queue had it been present when the handle was opened, but instead resolved to the default transmission queue, is not made invalid.

- A change to the *DefXmitQName* queue manager attribute. In this case all open handles that resolved to the previously named queue (that resolved to it only because it was the default transmission queue) are marked as invalid. Handles that resolved to this queue for other reasons are not affected.
- The *Shareability* queue attribute, if there are two or more handles that are currently providing OOINPS access for this queue, or for a queue that resolves to this queue. If so, *all* handles that are open for this queue, or for a queue that resolves to this queue, are marked as invalid, regardless of the open options.
- The *Usage* queue attribute, for all handles that are open for this queue, or for a queue that resolves to this queue, regardless of the open options.

When a handle is marked as invalid, all subsequent calls (other than MQCLOSE) using this handle fail with reason code RC2041; the application should issue an MQCLOSE call (using the original handle) and then reopen the queue. Any uncommitted updates against the old handle from previous successful calls can still be committed or backed out, as required by the application logic.

If changing an attribute will cause this to happen, a special "force" version of the command must be used.

5. The queue manager performs security checks when an MQOPEN call is issued, to verify that the user identifier under which the application is running has the appropriate level of authority before access is permitted. The authority check is made on the name of the object being opened, and not on the name, or names, resulting after a name has been resolved.

If the object being opened is a model queue, the queue manager performs a full security check against both the name of the model queue and the name of the dynamic queue that is created. If the resulting dynamic queue is then opened explicitly, a further resource security check is performed against the name of the dynamic queue.

6. A remote queue can be specified in one of two ways in the *OBJDSC* parameter of this call (see the *ODON* and *ODMN* fields described in “MQOD - Object descriptor” on page 1117):

- By specifying for *ODON* the name of a local definition of the remote queue. In this case, *ODMN* refers to the local queue manager, and can be specified as blanks.

The security validation performed by the local queue manager verifies that the user is authorized to open the local definition of the remote queue.

- By specifying for *ODON* the name of the remote queue as known to the remote queue manager. In this case, *ODMN* is the name of the remote queue manager.

The security validation performed by the local queue manager verifies that the user is authorized to send messages to the transmission queue resulting from the name resolution process.

In either case:

- No messages are sent by the local queue manager to the remote queue manager in order to check that the user is authorized to put messages on the queue.
- When a message arrives at the remote queue manager, the remote queue manager might reject it because the user originating the message is not authorized.

7. An MQOPEN call with the OOBROW option establishes a browse cursor, for use with MQGET calls that specify the object handle and one of the browse options. This allows the queue to be scanned without altering its contents. A message that has been found by browsing can later be removed from the queue by using the GMMUC option.

Multiple browse cursors can be active for a single application by issuing several MQOPEN requests for the same queue.

8. The following notes apply to the use of distribution lists.

- Fields in the MQOD structure must be set as follows when opening a distribution list:
 - *ODVER* must be *ODVER2* or greater.
 - *ODOT* must be *OTQ*.
 - *ODON* must be blank or the null string.
 - *ODMN* must be blank or the null string.
 - *ODREC* must be greater than zero.
 - One of *ODORO* and *ODORP* must be zero and the other nonzero.
 - No more than one of *ODRRO* and *ODRRP* can be nonzero.
 - There must be *ODREC* object records, addressed by either *ODORO* or *ODORP*. The object records must be set to the names of the destination queues to be opened.
 - If one of *ODRRO* and *ODRRP* is nonzero, there must be *ODREC* response records present. They are set by the queue manager if the call completes with reason code RC2136.

A version-2 MQOD can also be used to open a single queue that is not in a distribution list, by ensuring that *ODREC* is zero.

- Only the following open options are valid in the *OPTS* parameter:
 - *OOOUT*
 - *OOPAS**
 - *OOSSET**
 - *OOALTU*
 - *OOFIQ*

- The destination queues in the distribution list can be local, alias, or remote queues, but they cannot be model queues. If a model queue is specified, that queue fails to open, with reason code RC2057. However, this does not prevent other queues in the list being opened successfully.
- The completion code and reason code parameters are set as follows:
 - If the open operations for the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

For example, if every open succeeds, the completion code is set to CCOK and the reason code is RCNONE; if every open fails because none of the queues exists, the parameters are set to CCFAIL and RC2085.
 - If the open operations for the queues in the distribution list do not all succeed or fail in the same way:
 - The completion code parameter is set to CCWARN if at least one open succeeded, and to CCFAIL if all failed.
 - The reason code parameter is set to RC2136.
 - The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.
- When a distribution list has been opened successfully, the handle *HOBj* returned by the call can be used on subsequent MQPUT calls to put messages to queues in the distribution list, and on an MQCLOSE call to relinquish access to the distribution list. The only valid close option for a distribution list is CONONE.

The MQPUT1 call can also be used to put a message to a distribution list; the MQOD structure defining the queues in the list is specified as a parameter on that call.

- Each successfully opened destination in the distribution list counts as a *separate* handle when checking whether the application has exceeded the permitted maximum number of handles (see the *MaxHandles* queue manager attribute). This is true even when two or more of the destinations in the distribution list actually resolve to the same physical queue. If the MQOPEN or MQPUT1 call for a distribution list would cause the number of handles in use by the application to exceed *MaxHandles*, the call fails with reason code RC2017.
- Each destination that is opened successfully has the value of its *OpenOutputCount* attribute incremented by one. If two or more of the destinations in the distribution list actually resolve to the same physical queue, that queue has its *OpenOutputCount* attribute incremented by the number of destinations in the distribution list that resolve to that queue.
- Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.
- It is valid for a distribution list to contain only one destination.

9. The following notes apply to the use of cluster queues.

- When a cluster queue is opened for the first time, and the local queue manager is not a full repository queue manager, the local queue manager obtains information about the cluster queue from a full repository queue manager. When the network is busy, it may take several seconds for the local queue manager to receive the needed information from the repository queue manager. As a result, the application issuing the MQOPEN call might have to wait for up to 10 seconds before control returns from the MQOPEN call. If the local queue manager does not receive the needed information about the cluster queue within this time, the call fails with reason code RC2189.
- When a cluster queue is opened and there are multiple instances of the queue in the cluster, the instance actually opened depends on the options specified on the MQOPEN call:
 - If the options specified include any of the following:
 - OOBROW

- OOINPQ
- OOINPX
- OOINPS
- OOSSET


the instance of the cluster queue opened is required to be the local instance. If there is no local instance of the queue, the MQOPEN call fails.

- If the options specified include none of the above, but do include one or both of the following:
 - OOINQ
 - OOOOUT

the instance opened is the local instance if there is one, and a remote instance otherwise. The instance chosen by the queue manager can, however, be altered by a cluster workload exit (if there is one).

For more information about cluster queues, see [Cluster queues](#).

10. Applications started by a trigger monitor are passed the name of the queue that is associated with the application when the application is started. This queue name can be specified in the *OBJDSC* parameter to open the queue. See the description of the MQTMC structure for further details.
11. On IBM i, applications running in compatibility mode are connected automatically to the queue manager by the first MQOPEN call issued by the application (if the application has not already connected to the queue manager by using the MQCONN call).

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

Applications not running in compatibility mode must issue the MQCONN or MQCONNEX call to connect to the queue manager explicitly, before using the MQOPEN call to open an object.

12. When using the OORLOQ option, the local queue is already returned when either a local, alias, or model queue is opened, but this is not the case when, for example, a remote queue or a non-local cluster queue is opened; the ResolvedQName and ResolvedQMGrName are entered with the RemoteQName and RemoteQMGrName found in the remote queue definition, or similarly with the chosen remote cluster queue. If OORLOQ is specified when opening, for example, a remote queue, ResolvedQName will now be the transmission queue which messages will be put to. The ResolvedQMGrName will be entered with the name of the local queue manager hosting the transmission queue. If a user is authorized for browse, input or output on a queue, they have the required authority to specify this flag on the MQOPEN call. No special authority is needed.

Parameters

The MQOPEN call has the following parameters:

HCONN (10-digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

OBJDSC (MQOD) - input/output

Object descriptor.

This is a structure that identifies the object to be opened; see [“MQOD - Object descriptor”](#) on page 1117 for details.

If the *ODON* field in the *OBJDSC* parameter is the name of a model queue, a dynamic local queue is created with the attributes of the model queue; this happens irrespective of the open options specified by the *OPTS* parameter. Subsequent operations using the *HOBJ* returned by the *MQOPEN* call are performed on the new dynamic queue, and not on the model queue. This is true even for the *MQINQ* and *MQSET* calls. The name of the model queue in the *OBJDSC* parameter is replaced with the name of the dynamic queue created. The type of the dynamic queue is determined by the value of the *DefinitionType* attribute of the model queue (see [“Attributes for queues”](#) on page 1324). For information about the close options applicable to dynamic queues, see the description of the *MQCLOSE* call.

OPTS (10-digit signed integer) - input

Options that control the action of *MQOPEN*.

At least one of the following options must be specified:

- *OOBRW*
- *OOINP** (only one of these)
- *OOINQ*
- *OOOUT*
- *OOSET*
- *OORLQ*

Other options can be specified as required. If more than one option is required, the values can be added (do not add the same constant more than once). Combinations that are not valid are noted; all other combinations are valid. Only options that are applicable to the type of object specified by *OBJDSC* are allowed (see [Valid MQOPEN options for each queue type](#)).

Access options: The following options control the type of operations that can be performed on the object:

OOINPQ

Open queue to get messages using queue-defined default.

The queue is opened for use with subsequent *MQGET* calls. The type of access is either shared or exclusive, depending on the value of the *DefInputOpenOption* queue attribute; see [“Attributes for queues”](#) on page 1324 for details.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

OOINPS

Open queue to get messages with shared access.

The queue is opened for use with subsequent *MQGET* calls. The call can succeed if the queue is currently open by this or another application with *OOINPS*, but fails with reason code RC2042 if the queue is currently open with *OOINPX*.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

OOINPX

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent *MQGET* calls. The call fails with reason code RC2042 if the queue is currently open by this or another application for input of any type (*OOINPS* or *OOINPX*).

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

The following notes apply to these options:

- Only one of these options can be specified.
- An MQOPEN call with one of these options can succeed even if the *InhibitGet* queue attribute is set to QAGETI (although subsequent MQGET calls will fail while the attribute is set to this value).
- If the queue is defined as not being shareable (that is, the *Shareability* queue attribute has the value QANSHR), attempts to open the queue for shared access are treated as attempts to open the queue with exclusive access.
- If an alias queue is opened with one of these options, the test for exclusive use (or for whether another application has exclusive use) is against the base queue to which the alias resolves.
- These options are not valid if *ODMN* is the name of a queue manager alias; this is true even if the value of the *RemoteQMGrName* attribute in the local definition of a remote queue used for queue manager aliasing is the name of the local queue manager.

OOBRW

Open queue to browse messages.

The queue is opened for use with subsequent MQGET calls with one of the following options:

- GMBRWF
- GMBRWN
- GMBRWC

This is allowed even if the queue is currently open for OOINPX. An MQOPEN call with the OOBRW option establishes a browse cursor, and positions it logically before the first message on the queue; see the *GMOPT* field described in [“MQGMO - Get-message options” on page 1035](#) for further information.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects which are not queues. It is also not valid if *ODMN* is the name of a queue manager alias; this is true even if the value of the *RemoteQMGrName* attribute in the local definition of a remote queue used for queue manager aliasing is the name of the local queue manager.

OOOUT

Open queue to put messages, or a topic or topic string to publish messages.

The queue is opened for use with subsequent MQPUT calls.

An MQOPEN call with this option can succeed even if the *InhibitPut* queue attribute is set to QAPUTI (although subsequent MQPUT calls will fail while the attribute is set to this value).

This option is valid for all types of queue, including distribution lists and topics.

OOINQ

Open object to inquire attributes.

The queue, namelist, process definition, or queue manager is opened for use with subsequent MQINQ calls.

This option is valid for all types of object other than distribution lists. It is not valid if *ODMN* is the name of a queue manager alias; this is true even if the value of the *RemoteQMGrName* attribute in the local definition of a remote queue used for queue manager aliasing is the name of the local queue manager.

OOSSET

Open queue to set attributes.

The queue is opened for use with subsequent MQSET calls.

This option is valid for all types of queue other than distribution lists. It is not valid if *ODMN* is the name of a local definition of a remote queue; this is true even if the value of the *RemoteQMGrName* attribute in the local definition of a remote queue used for queue manager aliasing is the name of the local queue manager.

Binding options: The following options apply when the object being opened is a cluster queue; these options control the binding of the queue handle to an instance of the cluster queue:

OOBNDQ

Bind handle to destination when queue is opened.

This causes the local queue manager to bind the queue handle to an instance of the destination queue when the queue is opened. As a result, all messages put using this handle are sent to the same instance of the destination queue, and by the same route.

This option is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

OOBNDN

Do not bind to a specific destination.

This stops the local queue manager binding the queue handle to an instance of the destination queue. As a result, successive MQPUT calls using this handle may result in the messages being sent to *different* instances of the destination queue, or being sent to the same instance but by different routes. It also allows the instance selected to be changed later by the local queue manager, by a remote queue manager, or by a message channel agent (MCA), according to network conditions.

Note: Client and server applications which need to exchange a *series* of messages in order to complete a transaction should not use OOBNDN (or OOBNDQ when *DefBind* has the value BNDNOT), because successive messages in the series may be sent to different instances of the server application.

If OOBROW or one of the OOINP* options is specified for a cluster queue, the queue manager is forced to select the local instance of the cluster queue. As a result, the binding of the queue handle is fixed, even if OOBNDN is specified.

If OOINQ is specified with OOBNDN, successive MQINQ calls using that handle may inquire different instances of the cluster queue, although typically all of the instances have the same attribute values.

OOBNDN is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

OOBNDQ

Use default binding for queue.

This causes the local queue manager to bind the queue handle in the way defined by the *DefBind* queue attribute. The value of this attribute is either BNDOPN or BNDNOT.

OOBNDQ is the default if OOBNDQ and OOBNDN are not specified.

OOBNDQ is defined to aid program documentation. It is not intended that this option is used with either of the other two bind options, but because its value is zero such use cannot be detected.

Context options: The following options control the processing of message context:

OOSAVA

Save context when message retrieved.

Context information is associated with this queue handle. This information is set from the context of any message retrieved using this handle. For more information about message context, see [Message context](#) and [Controlling context information](#).

This context information can be passed to a message that is later put on a queue using the MQPUT or MQPUT1 calls. See the PMPASI and PMPASA options described in [“MQPMO - Put-message options” on page 1131](#).

Until a message has been successfully retrieved, context cannot be passed to a message being put on a queue.

A message retrieved using one of the GMBRW* browse options does not have its context information saved (although the context fields in the *MSGDSC* parameter are set after a browse).

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects which are not queues. One of the OOINP* options must be specified.

OOPASI

Allow identity context to be passed.

This allows the PMPASI option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity context information from an input queue that was opened with the OOSAVA option. For more information about message context, see [Message context](#) and [Controlling context information](#).

The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOPASA

Allow all context to be passed.

This allows the PMPASA option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity and origin context information from an input queue that was opened with the OOSAVA option. For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies OOPASI, which need not therefore be specified. The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOSETI

Allow identity context to be set.

This allows the PMSETI option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity context information contained in the *MSGDSC* parameter specified on the MQPUT or MQPUT1 call. For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies OOPASI, which need not therefore be specified. The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOSETA

Allow all context to be set.

This allows the PMSETA option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity and origin context information contained in the *MSGDSC* parameter specified on the MQPUT or MQPUT1 call. For more information about message context, see [Message context](#) and [Controlling context information](#).

This option implies the following options, which need not therefore be specified:

- OOPASI
- OOPASA
- OOSETI

The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

Other options: The following options control authorization checking, and what happens when the queue manager is quiescing:

OOALTU

Validate with specified user identifier.

This indicates that the *ODAU* field in the *OBJDSC* parameter contains a user identifier that is to be used to validate this MQOPEN call. The call can succeed only if this *ODAU* is authorized to open the object with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so. This does not apply to any context options specified, however, which are always checked against the user identifier under which the application is running.

This option is valid for all types of object.

OOFIQ

Fail if queue manager is quiescing.

This option forces the MQOPEN call to fail if the queue manager is in quiescing state.

This option is valid for all types of object.

OO RLQ

Enter the name of local queue that was opened.

This option specifies that the ResolvedQName in the MQOD structure (if available) should be entered with the name of the local queue which was opened. The ResolvedQMGrName will similarly be entered with the name of the local queue manager hosting the local queue.

Valid MQOPEN options for each queue type

Option	Alias (“1” on page 1288)	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
OOINPQ	✓	✓	-	-	-	-
OOINPS	✓	✓	-	-	-	-
OOINPX	✓	✓	-	-	-	-
OOBRW	✓	✓	-	-	-	-
OOOUT	✓	✓	✓	✓	✓	✓
OOINQ	✓	✓	“2” on page 1288	✓	-	-
OOSSET	✓	✓	“2” on page 1288	-	-	-
OOBNDQ (“3” on page 1288)	✓	✓	✓	✓	✓	-
OOBNDN (“3” on page 1288)	✓	✓	✓	✓	✓	-
OOBNDQ (“3” on page 1288)	✓	✓	✓	✓	✓	-
OOSAVA	✓	✓	-	-	-	-
OOPASI	✓	✓	✓	✓	✓	“5” on page 1288
OOPASA	✓	✓	✓	✓	✓	“5” on page 1288

Option	Alias (“1” on page 1288)	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
OOSSETI	✓	✓	✓	✓	✓	“5” on page 1288
OOSSETA	✓	✓	✓	✓	✓	“5” on page 1288
OOSALTU	✓	✓	✓	✓	✓	✓
OOSFIQ	✓	✓	✓	✓	✓	✓
OOSRLQ	✓	✓	✓	✓	-	-

Notes:

1. The validity of options for aliases depends on the validity of the option for the queue to which the alias resolves.
2. This option is valid only for the local definition of a remote queue.
3. This option can be specified for any queue type, but is ignored if the queue is not a cluster queue.
4. This attribute is ignored for a topic.
5. These attributes can be used with a topic, but only affect the context set for the retained message, not the context fields sent to any subscriber.

HOBj (10-digit signed integer) - output

Object handle.

This handle represents the access that has been established to the object. It must be specified on subsequent message queuing calls that operate on the object. It ceases to be valid when the MQCLOSE call is issued, or when the unit of processing that defines the scope of the handle terminates.

The scope of the handle is restricted to the smallest unit of parallel processing supported by the platform on which the application is running; the handle is not valid outside the unit of parallel processing from which the MQOPEN call was issued:

- On IBM i, the scope of the handle is the job issuing the call.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQOPEN(HCONN : OBJDSC : OPTS :
C                      HOBj : CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQOPEN          PR          EXTPROC('MQOPEN')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object descriptor
D OBJDSC          468A
D* Options that control the action of MQOPEN
D OPTS          10I 0 VALUE
D* Object handle
D HOBJ          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0

```

MQPUT - Put message

The MQPUT call puts a message on a queue, distribution list or to a topic. The queue, distribution list, or topic must already be open.

- [“Syntax” on page 1289](#)
- [“Usage notes” on page 1289](#)
 - [“Topics” on page 1289](#)
 - [“MQPUT and MQPUT1” on page 1290](#)
 - [“Destination queues” on page 1290](#)
 - [“Distribution lists” on page 1291](#)
 - [“Headers” on page 1292](#)
 - [“Buffer” on page 1293](#)
- [“Parameters” on page 1293](#)
- [“RPG Declaration” on page 1298](#)

Syntax

MQPUT (*HCONN*, *HOBJ*, *MSGDSC*, *PMO*, *BUFLEN*, *BUFFER*, *CMPCOD*, *REASON*)

Usage notes

Topics

The following notes apply to the use of topics:

1. When using MQPUT to publish messages on a topic, where one or more subscribers to that topic cannot be given the publication due to a problem with their subscriber queue (for example it is full), the Reason code returned to the MQPUT call and the delivery behavior is dependent on the setting of the PMSGDLV or NPMSGDLV attributes on the TOPIC. Note that delivery of a publication to the dead letter queue when RODLQ is specified, or discarding the message when RODISC is specified, is considered a successful delivery of the message. If none of the publications are delivered, the MQPUT will return with RC2502. This can happen in the following cases:
 - A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALL and any subscription (durable or not) has a queue which cannot receive the publication.
 - A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLDUR and a durable subscription has a queue which cannot receive the publication.

The MQPUT can return with RCNONE even though publications could not be delivered to some subscribers in the following cases:

- A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLAVAIL and any subscription, durable or not, has a queue which cannot receive the publication.
 - A message is published to a TOPIC with PMSGDLV or NPMSGDLV (depending on the persistence of the message) set to ALLDUR and a non-durable subscription has a queue which cannot receive the publication.
2. If there are no subscribers to the topic being used, the message published is not sent to any queue and is discarded. It does not make any difference whether this message is persistent or non-persistent, or whether it has unlimited expiry or some small expiry time, it is still discarded if there are no subscribers. The exception to this is if the message is to be retained, in which case, although it is not sent to any subscribers' queues, it is stored against the topic to be delivered to any new subscriptions or to any subscribers that ask for retained publications using MQSUBRQ.

MQPUT and MQPUT1

Both the MQPUT and MQPUT1 calls can be used to put messages on a queue; which call to use depends on the circumstances

- The MQPUT call should be used when multiple messages are to be placed on the *same* queue.

An MQOPEN call specifying the OOOUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.

- The MQPUT1 call should be used when only *one* message is to be put on a queue.

This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, minimizing the number of calls that must be issued.

Destination queues

If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if the following conditions are satisfied. Some conditions apply to both local and remote destination queues; other conditions apply only to remote destination queues.

Conditions for local and remote destination queues

- All of the MQPUT calls are within the same unit of work, or none of them is within a unit of work.

When messages are put onto a particular queue within a single unit of work, messages from other applications might be interspersed with the sequence of messages on the queue.

- All of the MQPUT calls are made using the same object handle *HOBJ*.

In some environments, message sequence is also preserved when different object handles are used, provided the calls are made from the same application. The meaning of "same application" is determined by the environment:

- On IBM i, the application is the job.

- The messages all have the same priority.

Additional conditions for remote destination queues

- There is only one path from the sending queue manager to the destination queue manager.

If there is a possibility that some messages in the sequence may go on a different path (for example, because of reconfiguration, traffic balancing, or path selection based on message size), the order of the messages at the destination queue manager cannot be guaranteed.

- Messages are not placed temporarily on dead-letter queues at the sending, intermediate, or destination queue managers.

If one or more of the messages is put temporarily on a dead-letter queue (for example, because a transmission queue or the destination queue is temporarily full), the messages can arrive on the destination queue out of sequence.

- The messages are either all persistent or all nonpersistent.

If a channel on the route between the sending and destination queue managers has its *CDNPM* attribute set to *NPFAST*, nonpersistent messages can jump ahead of persistent messages, resulting in the order of persistent messages relative to nonpersistent messages not being preserved. However, the order of persistent messages relative to each other, and of nonpersistent messages relative to each other, is preserved.

If these conditions are not satisfied, message groups can be used to preserve message order, but note that this requires both the sending and receiving applications to use the message-grouping support. For more information about message groups, see:

- *MDMFL* field in MQMD
- *PMLOGO* option in MQPMO
- *GMLOGO* option in MQGMO

Distribution lists

The following notes apply to the use of distribution lists.

1. Messages can be put to a distribution list using either a version-1 or a version-2 MQPMO. If a version-1 MQPMO is used (or a version-2 MQPMO with *PMREC* equal to zero), no put message records or response records can be provided by the application. This means that it will not be possible to identify the queues which encounter errors, if the message is sent successfully to some queues in the distribution list and not others.

If put message records or response records are provided by the application, the *PMVER* field must be set to *PMVER2*.

A version-2 MQPMO can also be used to send messages to a single queue that is not in a distribution list, by ensuring that *PMREC* is zero.

2. The completion code and reason code parameters are set as follows:

- If the puts to the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

For example, if every put succeeds, the completion code is set to *CCOK* and the reason code is *RCNONE*; if every put fails because all of the queues are inhibited for puts, the parameters are set to *CCFAIL* and *RC2051*.

- If the puts to the queues in the distribution list do not all succeed or fail in the same way:
 - The completion code parameter is set to *CCWARN* if at least one put succeeded, and to *CCFAIL* if all failed.
 - The reason code parameter is set to *RC2136*.
 - The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.

If the put to a destination fails because the open for that destination failed, the fields in the response record are set to *CCFAIL* and *RC2137*; that destination is included in *PMIDC*.

3. If a destination in the distribution list resolves to a local queue, the message is placed on that queue in normal form (that is, not as a distribution-list message). If more than one destination resolves to the same local queue, one message is placed on the queue for each such destination.

If a destination in the distribution list resolves to a remote queue, a message is placed on the appropriate transmission queue. Where several destinations resolve to the same transmission queue, a single distribution-list message containing those destinations may be placed on the transmission queue, even if those destinations were not adjacent in the list of destinations provided by the application. However, this can be done only if the transmission queue supports distribution-list messages (see the *DistLists* queue attribute described in [“Attributes for queues” on page 1324](#)).

If the transmission queue does not support distribution lists, one copy of the message in normal form is placed on the transmission queue for each destination that uses that transmission queue.

If a distribution list with the application message data is too large for a transmission queue, the distribution list message is split up into smaller distribution-list messages, each containing fewer destinations. If the application message data only just fits on the queue, distribution-list messages cannot be used at all, and the queue manager generates one copy of the message in normal form for each destination that uses that transmission queue.

If different destinations have different message priority or message persistence (this can occur when the application specifies PRQDEF or PEQDEF), the messages are not held in the same distribution-list message. Instead, the queue manager generates as many distribution-list messages as are necessary to accommodate the differing priority and persistence values.

4. A put to a distribution list might result in:

- A single distribution-list message, or
- A number of smaller distribution-list messages, or
- A mixture of distribution list messages and normal messages, or
- Normal messages only.

Which of the previous occurs depends on whether:

- The destinations in the list are local, remote, or a mixture.
- The destinations have the same message priority and message persistence.
- The transmission queues can hold distribution-list messages.
- The transmission queues' maximum message lengths are large enough to accommodate the message in distribution-list form.

However, regardless of which of the above occurs, each *physical* message resulting (that is, each normal message or distribution-list message resulting from the put) counts as only *one* message when:

- Checking whether the application has exceeded the permitted maximum number of messages in a unit of work (see the *MaxUncommittedMsgs* queue manager attribute).
- Checking whether the triggering conditions are satisfied.
- Incrementing queue depths and checking whether the queues' maximum queue depth would be exceeded.

5. Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.

Headers

If a message is put with one or more IBM MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. If the queue manager detects an error, the call fails with an appropriate reason code. The checks performed vary according to the particular structures that are present. In addition, the checks are performed only if a version-2 or later MQMD is used on the MQPUT or MQPUT1 call; the checks are not performed if a version-1 MQMD is used, even if an MQMDE is present at the start of the application message data.

The following IBM MQ header structures are validated completely by the queue manager: MQDH, MQMDE.

For other IBM MQ header structures, the queue manager performs some validation, but does not check every field. Structures that are not supported by the local queue manager, and structures following the first MQDLH in the message, are not validated.

In addition to general checks on the fields in IBM MQ structures, the following conditions must be satisfied:

- An IBM MQ structure must not be split over two or more segments - the structure must be entirely contained within one segment.
- The sum of the lengths of the structures in a PCF message must equal the length specified by the *BUFLen* parameter on the MQPUT or MQPUT1 call. A PCF message is a message that has one of the following format names:
 - FMADMN
 - FMEVNT
 - FMPCF
- IBM MQ structures must not be truncated, except in the following situations where truncated structures are permitted:
 - Messages which are report messages.
 - PCF messages.
 - Messages containing an MQDLH structure. (Structures *following* the first MQDLH can be truncated; structures preceding the MQDLH cannot.)

Buffer

The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter should be declared instead as a structure, or as a field in a physical file. This will increase the maximum length possible to approximately 32 KB.

Parameters

The MQPUT call has the following parameters:

HCONN (10-digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HOBJ (10-digit signed integer) - input

Object handle.

This handle represents the queue to which the message is added, or the topic to which the message is published. The value of *HOBJ* was returned by a previous MQOPEN call that specified the OOOUT option.

MSGDSC (MQMD) - input/output

Message descriptor.

This structure describes the attributes of the message being sent, and receives information about the message after the put request is complete. See [“MQMD - Message descriptor” on page 1068](#) for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure in order to specify values for the fields that exist in the version-2 MQMD but not the

version-1. The *MDFMT* field in the MQMD must be set to FMMDE to indicate that an MQMDE is present. See “MQMDE - Message descriptor extension” on page 1111 for more details.

PMO (MQPMO) - input/output

Options that control the action of MQPUT.

See “MQPMO - Put-message options” on page 1131 for details.

BUFLEN (10-digit signed integer) - input

Length of the message in *BUFFER*.

Zero is valid, and indicates that the message contains no application data. The upper limit for *BUFLEN* depends on various factors:

- If the destination queue is a shared queue, the upper limit is 63 KB (64 512 bytes).
- If the destination is a local queue or resolves to a local queue (but is not a shared queue), the upper limit depends on whether:
 - The local queue manager supports segmentation.
 - The sending application specifies the flag that allows the queue manager to segment the message. This flag is MFSEGA, and can be specified either in a version-2 MQMD, or in an MQMDE used with a version-1 MQMD.

If both of these conditions are satisfied, *BUFLEN* cannot exceed 999 999 999 minus the value of the *MDOFF* field in MQMD. The longest logical message that can be put is therefore 999 999 999 bytes (when *MDOFF* is zero). However, resource constraints imposed by the operating system or environment in which the application is running may result in a lower limit.

If one or both of the previously described conditions are not satisfied, *BUFLEN* cannot exceed the smaller of the queue's *MaxMsgLength* attribute and queue manager's *MaxMsgLength* attribute.

- If the destination is a remote queue or resolves to a remote queue, the conditions for local queues apply, *but at each queue manager through which the message must pass in order to reach the destination queue* ; in particular:
 1. The local transmission queue used to store the message temporarily at the local queue manager
 2. Intermediate transmission queues (if any) used to store the message at queue managers on the route between the local and destination queue managers
 3. The destination queue at the destination queue manager

The longest message that can be put is therefore governed by the most restrictive of these queues and queue managers.

When a message is on a transmission queue, additional information resides with the message data, and this reduces the amount of application data that can be carried. In this situation it is recommended that LNMHD bytes be subtracted from the *MaxMsgLength* values of the transmission queues when determining the limit for *BUFLEN*.

Note: Only failure to comply with condition 1 can be diagnosed synchronously (with reason code RC2030 or RC2031) when the message is put. If conditions 2 or 3 are not satisfied, the message is redirected to a dead-letter (undelivered-message) queue, either at an intermediate queue manager or at the destination queue manager. If this happens, a report message is generated if one was requested by the sender.

BUFFER (1-byte bit string x BUFLEN) - input

Message data.

This is a buffer containing the application data to be sent. The buffer should be aligned on a boundary appropriate to the nature of the data in the message. 4-byte alignment should be suitable for most messages (including messages containing MQ header structures), but some messages may require more stringent alignment. For example, a message containing a 64-bit binary integer might require 8-byte alignment.

If *BUFFER* contains character data, numeric data, or both, the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter should be set to the values appropriate to the data; this will enable the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All of the other parameters on the MQPUT call must be in the character set given by the *CodedCharSetId* queue manager attribute, and encoding of the local queue manager given by the ENNAT.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2104

(2104, X'838') Report option in message descriptor not recognized.

RC2136

(2136, X'858') Multiple reason codes returned.

If *CMPCOD* is CCFAIL:

RC2004

(2004, X'7D4') Buffer parameter not valid.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2013

(2013, X'7DD') Expiry time not valid.

RC2014

(2014, X'7DE') Feedback code not valid.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2024

(2024, X'7E8') No more messages can be handled within current unit of work.

RC2026

(2026, X'7EA') Message descriptor not valid.

RC2027

(2027, X'7EB') Missing reply-to queue.

RC2029
(2029, X'7ED') Message type in message descriptor not valid.

RC2030
(2030, X'7EE') Message length greater than maximum for queue.

RC2031
(2031, X'7EF') Message length greater than maximum for queue manager.

RC2039
(2039, X'7F7') Queue not open for output.

RC2041
(2041, X'7F9') Object definition changed since opened.

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2047
(2047, X'7FF') Persistence not valid.

RC2048
(2048, X'800') Queue does not support persistent messages.

RC2050
(2050, X'802') Message priority not valid.

RC2051
(2051, X'803') Put calls inhibited for the queue.

RC2052
(2052, X'804') Queue has been deleted.

RC2053
(2053, X'805') Queue already contains maximum number of messages.

RC2056
(2056, X'808') No space available on disk for queue.

RC2058
(2058, X'80A') Queue manager name not valid or not known.

RC2059
(2059, X'80B') Queue manager not available for connection.

RC2061
(2061, X'80D') Report options in message descriptor not valid.

RC2071
(2071, X'817') Insufficient storage available.

RC2072
(2072, X'818') Syncpoint support not available.

RC2093
(2093, X'82D') Queue not open for pass all context.

RC2094
(2094, X'82E') Queue not open for pass identity context.

RC2095
(2095, X'82F') Queue not open for set all context.

RC2096
(2096, X'830') Queue not open for set identity context.

RC2097
(2097, X'831') Queue handle referred to does not save context.

RC2098
(2098, X'832') Context not available for queue handle referred to.

RC2101
(2101, X'835') Object damaged.

RC2102
(2102, X'836') Insufficient system resources available.

RC2135
(2135, X'857') Distribution header structure not valid.

RC2136
(2136, X'858') Multiple reason codes returned.

RC2137
(2137, X'859') Object not opened successfully.

RC2149
(2149, X'865') PCF structures not valid.

RC2154
(2154, X'86A') Number of records present not valid.

RC2156
(2156, X'86C') Response records not valid.

RC2158
(2158, X'86E') Put message record flags not valid.

RC2159
(2159, X'86F') Put message records not valid.

RC2161
(2161, X'871') Queue manager quiescing.

RC2162
(2162, X'872') Queue manager shutting down.

RC2173
(2173, X'87D') Put-message options structure not valid.

RC2185
(2185, X'889') Inconsistent persistence specification.

RC2188
(2188, X'88C') Call rejected by cluster workload exit.

RC2189
(2189, X'88D') Cluster name resolution failed.

RC2195
(2195, X'893') Unexpected error occurred.

RC2219
(2219, X'8AB') MQI call reentered before previous call complete.

RC2241
(2241, X'8C1') Message group not complete.

RC2242
(2242, X'8C2') Logical message not complete.

RC2245
(2245, X'8C5') Inconsistent unit-of-work specification.

RC2248
(2248, X'8C8') Message descriptor extension not valid.

RC2249
(2249, X'8C9') Message flags not valid.

RC2250
(2250, X'8CA') Message sequence number not valid.

RC2251
(2251, X'8CB') Message segment offset not valid.

RC2252
(2252, X'8CC') Original length not valid.

RC2253

(2253, X'8CD') Length of data in message segment is zero.

RC2255

(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2257

(2257, X'8D1') Wrong version of MQMD supplied.

RC2258

(2258, X'8D2') Group identifier not valid.

RC2266

(2266, X'8DA') Cluster workload exit failed.

RC2269

(2269, X'8DD') Cluster resource error.

RC2270

(2270, X'8DE') No destination queues available.

RC2420

(2420) An MQPUT call was issued, but the message data contains an MQEPH structure that is not valid.

RC2479

(2479, X'9AF') Publication could not be retained.

RC2480

(2480, X'9B0') Target type has changed: the alias queue referred to a queue but now refers to a topic.

RC2502

(2502, X'9C6') Publication failed, and publication has not been delivered to any subscribers

RC2551

(2551, X'9F7') Specified selection string is not available.

RC2554

(2554, X'9FA') Message content could not be parsed to determine whether the message should be delivered to a subscriber with an extended message selector.

RPG Declaration

```

C*..1.....2.....3.....4.....5.....6.....7..
C                               CALLP      MQPUT(HCONN : HOBJ : MSGDSC : PMO :
C                               BUFLN : BUFFER : CMPCOD :
C                               REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQPUT          PR              EXTPROC('MQPUT')
D* Connection handle
D HCONN                10I 0 VALUE
D* Object handle
D HOBJ                10I 0 VALUE
D* Message descriptor
D MSGDSC                364A
D* Options that control the action of MQPUT
D PMO                  200A
D* Length of the message in Buffer
D BUFLN                10I 0 VALUE
D* Message data
D BUFFER              *   VALUE
D* Completion code
D CMPCOD                10I 0
D* Reason code qualifying CMPCOD
D REASON                10I 0

```

MQPUT1 - Put one message

The MQPUT1 call puts one message on a queue or distribution list, or to a topic. The queue, distribution list, or topic does not need to be open.

- [“Syntax” on page 1299](#)
- [“Usage notes” on page 1299](#)
- [“Parameters” on page 1300](#)
- [“RPG Declaration” on page 1305](#)

Syntax

MQPUT1 (*HCONN, OBJDSC, MSGDSC, PMO, BUFLN, BUFFER, CMPCOD, REASON*)

Usage notes

1. Both the MQPUT and MQPUT1 calls can be used to put messages on a queue; which call to use depends on the circumstances:
 - The MQPUT call should be used when multiple messages are to be placed on the *same* queue.
An MQOPEN call specifying the OOOUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.
 - The MQPUT1 call should be used when only *one* message is to be put on a queue.
This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, minimizing the number of calls that must be issued.
2. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved if certain conditions are satisfied. However, in most environments the MQPUT1 call does not satisfy these conditions, and so does not preserve message order. The MQPUT call must be used instead in these environments. See the usage notes in the description of the MQPUT call for details.
3. The MQPUT1 call can be used to put messages to distribution lists. For general information about this, see the usage notes for the MQOPEN and MQPUT calls.

The following differences apply when using the MQPUT1 call:

- a. If MQRN response records are provided by the application, they must be provided using the MQOD structure; they cannot be provided using the MQPMO structure.
 - b. The reason code RC2137 is never returned by MQPUT1 in the response records; if a queue fails to open, the response record for that queue contains the actual reason code resulting from the open operation.

If an open operation for a queue succeeds with a completion code of CCWARN, the completion code and reason code in the response record for that queue are replaced by the completion and reason codes resulting from the put operation.

As with the MQOPEN and MQPUT calls, the queue manager sets the response records (if provided) only when the outcome of the call is not the same for all queues in the distribution list; this is indicated by the call completing with reason code RC2136.
4. If the MQPUT1 call is used to put a message on a cluster queue, the call behaves as though OOBNDN had been specified on the MQOPEN call.
 5. If a message is put with one or more IBM MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. For more information about this, see the usage notes for the MQPUT call.

6. If more than one of the warning situations arise (see the *CMPCOD* parameter), the reason code returned is the *first* one in the following list that applies:
 - a. RC2136
 - b. RC2242
 - c. RC2241
 - d. RC2049 or RC2104
7. The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter should be declared instead as a structure, or as a field in a physical file. This will increase the maximum length possible to approximately 32 KB.

Parameters

The MQPUT1 call has the following parameters:

HCONN (10-digit signed integer) - input


Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

OBJDSC (MQOD) - input/output

Object descriptor.

This is a structure which identifies the queue to which the message is added. See [“MQOD - Object descriptor” on page 1117](#) for details.

The user must be authorized to open the queue for output. The queue must **not** be a model queue.

MSGDSC (MQMD) - input/output

Message descriptor.

This structure describes the attributes of the message being sent, and receives feedback information after the put request is complete. See [“MQMD - Message descriptor” on page 1068](#) for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure in order to specify values for the fields that exist in the version-2 MQMD but not the version-1. The *MDFMT* field in the MQMD must be set to FMMDE to indicate that an MQMDE is present. See [“MQMDE - Message descriptor extension” on page 1111](#) for more details.

PMO (MQPMO) - input/output

Options that control the action of MQPUT1.

See [“MQPMO - Put-message options” on page 1131](#) for details.

BUFLEN (10-digit signed integer) - input

Length of the message in *BUFFER*.

Zero is valid, and indicates that the message contains no application data. The upper limit depends on various factors; see the description of the *BUFLEN* parameter of the MQPUT call for further details.

BUFFER (1-byte bit string x BUFLen) - input

Message data.

This is a buffer containing the application message data to be sent. The buffer should be aligned on a boundary appropriate to the nature of the data in the message. 4-byte alignment should be suitable for most messages (including messages containing IBM MQ header structures), but some messages may require more stringent alignment. For example, a message containing a 64-bit binary integer might require 8-byte alignment.

If *BUFFER* contains character data, numeric data, or both, the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter should be set to the values appropriate to the data; this will enable the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All of the other parameters on the MQPUT1 call must be in the character set given by the *CodedCharSetId* queue manager attribute and encoding of the local queue manager given by ENNAT.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2104

(2104, X'838') Report option in message descriptor not recognized.

RC2136

(2136, X'858') Multiple reason codes returned.

RC2049

(2049, X'801') Message Priority exceeds maximum value supported.

RC2241

(2241, X'8C1') Message group not complete.

RC2242

(2242, X'8C2') Logical message not complete.

If *CMPCOD* is CCFAIL:

RC2001

(2001, X'7D1') Alias base queue not a valid type.

RC2004

(2004, X'7D4') Buffer parameter not valid.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2009

(2009, X'7D9') Connection to queue manager lost.

- RC2013**
(2013, X'7DD') Expiry time not valid.
- RC2014**
(2014, X'7DE') Feedback code not valid.
- RC2017**
(2017, X'7E1') No more handles available.
- RC2018**
(2018, X'7E2') Connection handle not valid.
- RC2024**
(2024, X'7E8') No more messages can be handled within current unit of work.
- RC2026**
(2026, X'7EA') Message descriptor not valid.
- RC2027**
(2027, X'7EB') Missing reply-to queue.
- RC2029**
(2029, X'7ED') Message type in message descriptor not valid.
- RC2030**
(2030, X'7EE') Message length greater than maximum for queue.
- RC2031**
(2031, X'7EF') Message length greater than maximum for queue manager.
- RC2035**
(2035, X'7F3') Not authorized for access.
- RC2042**
(2042, X'7FA') Object already open with conflicting options.
- RC2043**
(2043, X'7FB') Object type not valid.
- RC2044**
(2044, X'7FC') Object descriptor structure not valid.
- RC2046**
(2046, X'7FE') Options not valid or not consistent.
- RC2047**
(2047, X'7FF') Persistence not valid.
- RC2048**
(2048, X'800') Queue does not support persistent messages.
- RC2050**
(2050, X'802') Message priority not valid.
- RC2051**
(2051, X'803') Put calls inhibited for the queue.
- RC2052**
(2052, X'804') Queue has been deleted.
- RC2053**
(2053, X'805') Queue already contains maximum number of messages.
- RC2056**
(2056, X'808') No space available on disk for queue.
- RC2057**
(2057, X'809') Queue type not valid.
- RC2058**
(2058, X'80A') Queue manager name not valid or not known.
- RC2059**
(2059, X'80B') Queue manager not available for connection.

RC2061
(2061, X'80D') Report options in message descriptor not valid.

RC2063
(2063, X'80F') Security error occurred.

RC2071
(2071, X'817') Insufficient storage available.

RC2072
(2072, X'818') Syncpoint support not available.

RC2082
(2082, X'822') Unknown alias base queue.

RC2085
(2085, X'825') Unknown object name.

RC2086
(2086, X'826') Unknown object queue manager.

RC2087
(2087, X'827') Unknown remote queue manager.

RC2091
(2091, X'82B') Transmission queue not local.

RC2092
(2092, X'82C') Transmission queue with wrong usage.

RC2097
(2097, X'831') Queue handle referred to does not save context.

RC2098
(2098, X'832') Context not available for queue handle referred to.

RC2101
(2101, X'835') Object damaged.

RC2102
(2102, X'836') Insufficient system resources available.

RC2135
(2135, X'857') Distribution header structure not valid.

RC2136
(2136, X'858') Multiple reason codes returned.

RC2149
(2149, X'865') PCF structures not valid.

RC2154
(2154, X'86A') Number of records present not valid.

RC2155
(2155, X'86B') Object records not valid.

RC2156
(2156, X'86C') Response records not valid.

RC2158
(2158, X'86E') Put message record flags not valid.

RC2159
(2159, X'86F') Put message records not valid.

RC2161
(2161, X'871') Queue manager quiescing.

RC2162
(2162, X'872') Queue manager shutting down.

RC2173
(2173, X'87D') Put-message options structure not valid.

RC2184

(2184, X'888') Remote queue name not valid.

RC2188

(2188, X'88C') Call rejected by cluster workload exit.

RC2189

(2189, X'88D') Cluster name resolution failed.

RC2195

(2195, X'893') Unexpected error occurred.

RC2196

(2196, X'894') Unknown transmission queue.

RC2197

(2197, X'895') Unknown default transmission queue.

RC2198

(2198, X'896') Default transmission queue not local.

RC2199

(2199, X'897') Default transmission queue usage error.

RC2258

(2258, X'8D2') Group identifier not valid.

RC2248

(2248, X'8C8') Message descriptor extension not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2249

(2249, X'8C9') Message flags not valid.

RC2250

(2250, X'8CA') Message sequence number not valid.

RC2251

(2251, X'8CB') Message segment offset not valid.

RC2252

(2252, X'8CC') Original length not valid.

RC2253

(2253, X'8CD') Length of data in message segment is zero.

RC2255

(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2257

(2257, X'8D1') Wrong version of MQMD supplied.

RC2266

(2266, X'8DA') Cluster workload exit failed.

RC2269

(2269, X'8DD') Cluster resource error.

RC2270

(2270, X'8DE') No destination queues available.

RC2420

(2420) An MQPUT1 call was issued, but the message data contains an MQEPH structure that is not valid.

RC2551

(2551, X'9F7') Specified selection string is not available.

RC2554

(2554, X'9FA') Message content could not be parsed to determine whether the message should be delivered to a subscriber with an extended message selector.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..  
C          CALLP      MQPUT1(HCONN : OBJDSC : MSGDSC :  
C                      PMO : BUFLN : BUFFER :  
C                      CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..  
DMQPUT1      PR          EXTPROC('MQPUT1')  
D* Connection handle  
D HCONN          10I 0 VALUE  
D* Object descriptor  
D OBJDSC          468A  
D* Message descriptor  
D MSGDSC          364A  
D* Options that control the action of MQPUT1  
D PMO            200A  
D* Length of the message in BUFFER  
D BUFLN          10I 0 VALUE  
D* Message data  
D BUFFER          *   VALUE  
D* Completion code  
D CMPCOD          10I 0  
D* Reason code qualifying CMPCOD  
D REASON          10I 0
```

MQSET - Set object attributes


The MQSET call is used to change the attributes of an object represented by a handle. The object must be a queue.

- [“Syntax” on page 1305](#)
- [“Usage notes” on page 1305](#)
- [“Parameters” on page 1306](#)
- [“RPG Declaration” on page 1309](#)

Syntax

MQSET (*HCONN*, *HOBJ*, *SELCNT*, *SELS*, *IACNT*, *INTATR*, *CALEN*, *CHRATR*, *CMPCOD*, *REASON*)

Usage notes

1. Using this call, the application can specify an array of integer attributes, or a collection of character attribute strings, or both. If no errors occur, the attributes specified are all set simultaneously. If an error occurs (for example, if a selector is not valid, or an attempt is made to set an attribute to a value that is not valid), the call fails and no attributes are set.
2. The values of attributes can be determined using the MQINQ call ; see [“MQINQ - Inquire about object attributes” on page 1261](#) for details.
Note: Not all attributes with values that can be inquired upon using the MQINQ call can have their values changed using the MQSET call. For example, no process-object or queue manager attributes can be set with this call.
3. Attribute changes are preserved across restarts of the queue manager (other than alterations to temporary dynamic queues, which do not survive restarts of the queue manager).
4. You cannot change the attributes of a model queue using the MQSET call. However, if you open a model queue using the MQOPEN call with the MQOO_SET option, you can use the MQSET call to set the attributes of the dynamic local queue that is created by the MQOPEN call.
5. If the object being set is a cluster queue, there must be a local instance of the cluster queue for the open to succeed.

IBM i For more information about object attributes, see:

- “Attributes for queues” on page 1324
- “Attributes for namelists” on page 1352
- “Attributes for process definitions” on page 1353
- “Attributes for the queue manager” on page 1355

Parameters

The MQSET call has the following parameters:

HCONN (10-digit signed integer) - input

Connection handle.

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important: **IBM i** If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HOBJ (10-digit signed integer) - input

Object handle.

This handle represents the queue object with attributes that are to be set. The handle was returned by a previous MQOPEN call that specified the OOSSET option.

SELCNT (10-digit signed integer) - input

Count of selectors.

This is the count of selectors that are supplied in the *SELS* array. It is the number of attributes that are to be set. Zero is a valid value. The maximum number allowed is 256.

SELS (10-digit signed integer x SELCNT) - input

Array of attribute selectors.

This is an array of *SELCNT* attribute selectors; each selector identifies an attribute (integer or character) with a value that is to be set.

Each selector must be valid for the type of queue that *HOBJ* represents. Only certain IA* and CA* values are allowed; these values are listed later in this section.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (IA* selectors) must be specified in *INTATR* in the same order in which these selectors occur in *SELS*. Attribute values that correspond to character attribute selectors (CA* selectors) must be specified in *CHRATR* in the same order in which those selectors occur. IA* selectors can be interleaved with the CA* selectors; only the relative order within each type is important.

It is not an error to specify the same selector more than once; if this is done, the last value specified for a particular selector is the one that takes effect.

Note:

1. The integer and character attribute selectors are allocated within two different ranges; the IA* selectors reside within the range IAFRST through IALAST, and the CA* selectors within the range CAFRST through CALAST.

For each range, the constants IALSTU and CALSTU define the highest value that the queue manager will accept.

2. If all the IA* selectors occur first, the same element numbers can be used to address corresponding elements in the *SELS* and *INTATR* arrays.

The attributes that can be set are listed in the following table. No other attributes can be set using this call. For the CA* attribute selectors, the constant that defines the length in bytes of the string that is required in *CHRATR* is provided in parentheses.

Table 625. MQSET attribute selectors for queues		
Selector	Description	Note
CATRGD	Trigger data (LNTRGD).	“2” on page 1307
IADIST	Distribution list support.	“1” on page 1307
IAIGET	Whether get operations are allowed.	
IAIPUT	Whether put operations are allowed.	
IATRGC	Trigger control.	“2” on page 1307
IATRGD	Trigger depth.	“2” on page 1307
IATRGP	Threshold message priority for triggers.	“2” on page 1307
IATRGT	Trigger type.	“2” on page 1307

Notes:

1. Supported only on AIX, HP-UX, IBM i, Solaris, Windows, plus IBM MQ clients connected to these systems.
2. Not supported on VSE/ESA.

IACNT (10-digit signed integer) - input

Count of integer attributes.

This is the number of elements in the *INTATR* array, and must be at least the number of IA* selectors in the *SELS* parameter. Zero is a valid value if there are none.

INTATR (10-digit signed integer x IACNT) - input

Array of integer attributes.

This is an array of *IACNT* integer attribute values. These attribute values must be in the same order as the IA* selectors in the *SELS* array.

CALEN (10-digit signed integer) - input

Length of character attributes buffer.

This is the length in bytes of the *CHRATR* parameter, and must be at least the sum of the lengths of the character attributes specified in the *SELS* array. Zero is a valid value if there are no CA* selectors in *SELS*.

CHRATR (1-byte character string x CALEN) - input

Character attributes.

This is the buffer containing the character attribute values, concatenated together. The length of the buffer is given by the *CALEN* parameter.

The characters attributes must be specified in the same order as the CA* selectors in the *SELS* array. The length of each character attribute is fixed (see *SELS*). If the value to be set for an attribute contains fewer nonblank characters than the defined length of the attribute, the value in *CHRATR* must be padded to the right with blanks to make the attribute value match the defined length of the attribute.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2006

(2006, X'7D6') Length of character attributes not valid.

RC2007

(2007, X'7D7') Character attributes string not valid.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2020

(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.

RC2021

(2021, X'7E5') Count of integer attributes not valid.

RC2023

(2023, X'7E7') Integer attributes array not valid.

RC2040

(2040, X'7F8') Queue not open for set.

RC2041

(2041, X'7F9') Object definition changed since opened.

RC2101

(2101, X'835') Object damaged.

RC2052

(2052, X'804') Queue has been deleted.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2065

(2065, X'811') Count of selectors not valid.

RC2067

(2067, X'813') Attribute selector not valid.

RC2066

(2066, X'812') Count of selectors too large.

RC2071

(2071, X'817') Insufficient storage available.

RC2075

(2075, X'81B') Value for trigger-control attribute not valid.

RC2076

(2076, X'81C') Value for trigger-depth attribute not valid.

RC2077

(2077, X'81D') Value for trigger-message-priority attribute not valid.

RC2078

(2078, X'81E') Value for trigger-type attribute not valid.

RC2195

(2195, X'893') Unexpected error occurred.

RPG Declaration

```

C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQSET(HCONN : HOBJ : SELCNT :
C                      SELS(1) : IACNT : INTATR(1) :
C                      CALEN : CHRATR : CMPCOD :
C                      REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQSET      PR          EXTPROC('MQSET')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object handle
D HOBJ          10I 0 VALUE
D* Count of selectors
D SELCNT        10I 0 VALUE
D* Array of attribute selectors
D SELS          10I 0
D* Count of integer attributes
D IACNT          10I 0 VALUE
D* Array of integer attributes
D INTATR        10I 0
D* Length of character attributes buffer
D CALEN          10I 0 VALUE
D* Character attributes
D CHRATR          *   VALUE
D* Completion code
D CMPCOD          10I 0

```

MQSETMP - Set message handle property

The MQSETMP call sets or modifies a property of a message handle.

- [“Syntax” on page 1310](#)
- [“Usage notes” on page 1310](#)
- [“Parameters” on page 1311](#)
- [“RPG Declaration” on page 1314](#)

Syntax

MQSETMP (*Hconn*, *Hmsg*, *SetPropOpts*, *Name*, *PropDesc*, *Type*, *ValueLength*, *Value*, *CompCode*, *Reason*)

Usage notes

- You can use this call only when the queue manager itself coordinates the unit of work. This can be:
 - A local unit of work, where the changes affect only IBM MQ resources.
 - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting IBM MQ resources.

For further details about local and global units of work, see [“MQBEGIN - Begin unit of work” on page 1209](#).

- In environments where the queue manager does not coordinate the unit of work, use the appropriate back-out call instead of MQBACK. The environment might also support an implicit back out caused by the application terminating abnormally.
 - On z/OS, use the following calls:
 - Batch programs (including IMS batch DL/I programs) can use the MQBACK call if the unit of work affects only IBM MQ resources. However, if the unit of work affects both IBM MQ resources and resources belonging to other resource managers (for example, Db2), use the SRRBACK call provided by the z/OS Recoverable Resource Service (RRS). The SRRBACK call backs out changes to resources belonging to the resource managers that have been enabled for RRS coordination.
 - CICS applications must use the EXEC CICS SYNCPOINT ROLLBACK command to back out the unit of work. Do not use the MQBACK call for CICS applications.
 - IMS applications (other than batch DL/I programs) must use IMS calls such as ROLB to back out the unit of work. Do not use the MQBACK call for IMS applications (other than batch DL/I programs).
 - On IBM i, use this call for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
- If an application ends with uncommitted changes in a unit of work, the disposition of those changes depends on whether the application ends normally or abnormally. See the usage notes in [“MQDISC - Disconnect queue manager” on page 1246](#) for further details.
- When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
 - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.
 - Whether the message is part of a unit of work.
 - For the MQPUT call: whether the message is persistent or nonpersistent.

The queue manager keeps three sets of group and segment information, one set for each of the following:

- The last successful MQPUT call (this can be part of a unit of work).
- The last successful MQGET call that removed a message from the queue (this can be part of a unit of work).
- The last successful MQGET call that browsed a message on the queue (this cannot be part of a unit of work).

If the application puts or gets the messages as part of a unit of work, and the application then decides to back out the unit of work, the group and segment information is restored to the value that it had previously:

- The information associated with the MQPUT call is restored to the value that it had before the first successful MQPUT call for that queue handle in the current unit of work.
- The information associated with the MQGET call is restored to the value that it had before the first successful MQGET call for that queue handle in the current unit of work.

Queues that were updated by the application after the unit of work started, but outside the scope of the unit of work, do not have their group and segment information restored if the unit of work is backed out.

Restoring the group and segment information to its previous value when a unit of work is backed out allows the application to spread a large message group or large logical message consisting of many segments across several units of work, and to restart at the correct point in the message group or logical message if one of the units of work fails.

Using several units of work might be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point if a system failure occurs.

For details of how to restart at the correct point after a system failure, see the PMLOGO option described in [PMOPT \(10 digit signed integer\)](#), and the GMLOGO option described in [GMOPT \(10 digit signed integer\)](#).

The remaining usage notes apply only when the queue manager coordinates the units of work:

- A unit of work has the same scope as a connection handle. All IBM MQ calls that affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See [HCONN \(10-digit signed integer\) - output](#) for information about the scope of connection handles.
- Only messages that were put or retrieved as part of the current unit of work are affected by this call.
- A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but that never issues a commit or backout call, can fill queues with messages that are not available to other applications. To guard against this possibility, the administrator must set the *MaxUncommittedMsgs* queue-manager attribute to a value that is low enough to prevent runaway applications filling the queues, but high enough to allow the expected messaging applications to work correctly.

Parameters

The MQSETMP call has the following parameters:

HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager.

The value must match the connection handle that was used to create the message handle specified in the *HMSG* parameter.

If the message handle was created using HCUNAS, a valid connection must be established on the thread setting a property of the message handle, otherwise the call fails with reason code RC2009 .

HMSG (20-digit signed integer) - input

This is the message handle to be modified. The value was returned by a previous MQCRTMH call.

SETOPT (MQSMPO) - input

Control how message properties are set.

This structure allows applications to specify options that control how message properties are set. The structure is an input parameter on the MQSETMP call. See [MQSMPO](#) for further information.

PRNAME (MQCHARV) - input

This is the name of the property to set.

See [Property names](#) and [Property name restrictions](#) for further information about the use of property names.

PRPDSC (MQPD) - input/output

This structure is used to define the attributes of a property, including:

- what happens if the property is not supported
- what message context the property belongs to
- what messages the property is copied into as it flows

See [MQPD](#) for further information about this structure.

TYPE (10 digit signed integer) - input

The data type of the property being set. It can be one of the following:

TYPBOL

A boolean. *ValueLength* must be 4.

TYPBST

A byte string. *ValueLength* must be zero or greater.

TYPI8

An 8 bit signed integer. *ValueLength* must be 1.

TYPI16

A 16 bit signed integer. *ValueLength* must be 2.

TYPI32

A 32 bit signed integer. *ValueLength* must be 4.

TYPI64

A 64 bit signed integer. *ValueLength* must be 8.

TYPF32

A 32 bit floating-point number. *ValueLength* must be 4.

TYPF64

A 64 bit floating-point number. *ValueLength* must be 8.

TYPSTR

A character string. *ValueLength* must be zero or greater, or the special value VLNULL.

TYPNUL

The property exists but has a null value. *ValueLength* must be zero.

VALLEN (10-digit signed integer) - input

The length in bytes of the property value in the *Value* parameter.

Zero is valid only for null values or for strings or byte strings. Zero indicates that the property exists but that the value contains no characters or bytes.

The value must be greater than or equal to zero or the following special value if the *Type* parameter has TYPSTR set:

VLNULL

The value is delimited by the first null encountered in the string. The null is not included as part of the string. This value is invalid if TYPSTR is not also set.

Note: The null character used to terminate a string if VLNULL is set is a null from the character set of the Value.

VALUE (1-byte bit string x VALLEN) - input

The value of the property to be set. The buffer must be aligned on a boundary appropriate to the nature of the data in the value.

In the C programming language, the parameter is declared as a pointer-to-void; the address of any type of data can be specified as the parameter.

If *ValueLength* is zero, *Value* is not referred to. In this case, the parameter address passed by programs written in C or System/390 assembler can be null.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2421

(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CMPCOD* is CCFAIL:

RC2204

(2204, X'089C') Adapter not available.

RC2130

(2130, X'852') Unable to load adapter service module.

RC2157

(2157, X'86D') Primary and home ASIDs differ.

RC2004

(2004, X'07D4') Value parameter not valid.

RC2005

(2005, X'07D5') Value length parameter not valid.

RC2219

(2219, X'08AB') MQI call entered before previous call completed.

RC2460

(2460, X'099C') Message handle pointer not valid.

RC2499

(2499, X'09C3') Message handle already in use.

RC2046

(2046, X'07FE') Options not valid or not consistent.

RC2482

(2482, X'09B2') Property descriptor structure not valid.

RC2442

(2442, X'098A') Invalid property name.

RC2473

(2473, X'09A9') Invalid property data type.

RC2472

(2472, X'09A8') Number format error encountered in value data.

RC2463

(2463, X'099F') Set message property options structure not valid.

RC2111

(2111, X'083F') Property name coded character set identifier not valid.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See [“Return codes for IBM i \(ILE RPG\)” on page 1381](#) for more details.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQSETMP(HCONN : HMSG : SETOPT :
                                           PRNAME : PRPDSC :
                                           TYPE : VALLEN : VALUE :
                                           CMPCOD : REASON)

```

The prototype definition for the call is:

```

DMQSETMP      PR      EXTPROC('MQSETMP')
D* Connection handle
D HCONN              10I 0 VALUE
D* Message handle
D HMSG              10I 0 VALUE
D* Options that control the action of MQSETMP
D SETOPT            20A
D* Property name
D PRNAME            32A
D* Property descriptor
D PRPDSC            24A
D* Property data type
D TYPE              10I 0 VALUE
D* Length of the Value area
D VALLEN            10I 0 VALUE
D* Property value
D VALUE              *   VALUE
D* Completion code
D CMPCOD            10I 0
D* Reason code qualifying CompCode
D REASON            10I 0

```

MQSTAT - Retrieve status information

Use the MQSTAT call to retrieve status information. The type of status information returned is determined by the STYPE value specified on the call.

- [“Syntax” on page 1314](#)
- [“Usage notes” on page 1315](#)
- [“Parameters” on page 1315](#)
- [“RPG Declaration” on page 1316](#)

Syntax

MQSTAT (HCONN, STYPE, STAT, CMPCOD, REASON)

Usage notes

1. A call to MQSTAT specifying a type of STATAPT returns information about previous asynchronous MQPUT and MQPUT1 operations. The MQSTAT structure passed on the call is completed with the first recorded asynchronous warning or error information for that connection. If further errors or warnings follow the first, they do not normally alter these values. However, if an error occurs with a completion code of CCWARN, a subsequent failure with a completion code of CCFAIL is returned instead.
2. If no errors have occurred since the connection was established or since the last call to MQSTAT then a CMPCOD of CCOK and REASON of RCNONE are returned.
3. Counts of the number of asynchronous calls that have been processed under the connection handle are returned by using three counters; STSPSC, STSPWC, and STSPFC. These counters are incremented by the queue manager each time an asynchronous operation is processed successfully, has a warning, or fails (note that for accounting purposes a put to a distribution list counts once per destination queue rather than once per distribution list).
4. A successful call to MQSTAT results in any previous error information or counts being reset.

Parameters

The MQSTAT call has the following parameters:

Hconn (MQHCONN) - input

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNEX call.

STYPE (10-digit signed integer) - input

Type of status information being requested. The only valid value is:

STATAPT

Return information about previous asynchronous put operations.

STS (MQSTS) - input/output

Status information structure. See [“MQSTS - Status reporting structure” on page 1186](#) for details.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2374

(2374, X'946') API exit failed

RC2183

(2183, X'887') Unable to load API exit.

RC2219

(2219, X'8AB') MQI call entered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2203

(2203, X'89B') Connection shutting down.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2162

(2162, X'872') Queue manager stopping

RC2102

(2102, X'836') Insufficient system resources available.

RC2430

(2430, X'97E') Error with MQSTAT type.

RC2071

(2071, X'817') Insufficient storage available.

RC2424

(2424, X'978') Error with MQSTS structure

RC2195

(2195, X'893') Unexpected error occurred.

RC2298

(2298, X'8FA') The function requested is not available in the current environment.

For detailed information about these codes, see:

- [Messages and reason codes](#)

RPG Declaration

```

C*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
C          CALLP      MQSTAT(HCONN : ETYPE : ERR :
C                      CMPCOD : REASON)

```

The prototype definition for the call is:

```

D.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
DMQSTAT      PR          EXTPROC('MQSTAT')
D* Connection handle
D HCONN              10I 0 VALUE
D* Status information type
D STYPE              10I 0 VALUE
D* Status information
D STATUS            296A
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CompCode
D REASON              10I 0

```

MQSUB - Register Subscription

The MQSUB call registers the applications subscription to a particular topic.

- [“Syntax” on page 1316](#)
- [“Usage notes” on page 1317](#)
- [“Parameters” on page 1318](#)
- [“RPG Declaration” on page 1322](#)

Syntax

MQSUB (*HCONN*, *SUBDSC*, *HOBJ*, *HSUB*, *CMPCOD*, *REASON*)

Usage notes

- The subscription is made to a topic, named either using the short name of a pre-defined topic object, the full name of the topic string, or it is formed by the concatenation of two parts, as described in [“Using topic strings”](#) on page 554.
- The queue manager performs security checks when an MQSUB call is issued, to verify that the user identifier under which the application is running has the appropriate level of authority before access is permitted. The appropriate topic object is located either by a short name being provided in the call, or the nearest short name object in the topic hierarchy being found if a long name is provided. An authority check is made on this topic object to ensure authority to subscribe is set and on the destination queue to ensure that authority for output is set. If the SDMAN option is used, this means that an authority check is made on the managed queue name associated with this topic object, and if a non-managed queue is provided, this means that an authority check is made on the queue represented by the *HOBJ* parameter.
- The *HOBJ* returned on the MQSUB call when the SOMAN option is used, can be inquired in order to find out attributes such as the Backout threshold and the Excessive backout requeue name. You can also inquire the name of the managed queue, but you should not attempt to directly open this queue.
- Subscriptions can be grouped allowing only a single publication to be delivered to the group of subscriptions even where more than one of the group matched the publication. Subscriptions are grouped using the SOGRP option and in order to group subscriptions they must:
 - be using the same named queue (that is not using the SOMAN option) on the same queue manager - represented by the *HOBJ* parameter on the MQSUB call
 - share the same *SDCID*
 - be of the same *SDSL*

These attributes define the set of subscriptions considered to be in the group, and are also the attributes that cannot be altered if a subscription is grouped. Alteration of *SDSL* results in RC2512, and alteration of any of the others (which can be changed if a subscription is not grouped) results in RC2515.

- Fields in the MQSD are completed on return from an MQSUB call which uses the SORES option. The MQSD returned can be passed directly into an MQSUB call which uses the SOALT option with any changes you need to make to the subscription applied to the MQSD. Some fields have special considerations as noted in the table.

MQSD output from MQSUB	
Field name in MQSD	Special considerations
Access or creation options	None of these options are set on return from the MQSUB call. If you later reuse the MQSD in an MQSUB call the option you require must be explicitly set.
Durability options, Destination options, Registration Options & Wildcard options	These options will be set as appropriate
Publication options	These options will be set as appropriate, except for SONEWP which is only applicable to SOCRE.
Other options	These options are unchanged on return from an MQSUB call. They control how the API call is issued and are not stored with the subscription. They must be set as required on any subsequent MQSUB call reusing the MQSD.
ObjectName	This input only field is unchanged on return from an MQSUB call.

MQSD output from MQSUB (<i>continued</i>)	
Field name in MQSD	Special considerations
ObjectString	This input only field is unchanged on return from an MQSUB call. The Full topic name used is returned in the <i>SDRO</i> field, if a buffer is provided.
AlternateUserId and AlternateSecurityId	These input only fields are unchanged on return from an MQSUB call. They control how the API call is issued and are not stored with the subscription. They must set as required on any subsequent MQSUB call reusing the MQSD.
SubExpiry	On return from an MQSUB call using the SORES option this field will be set to the original expiry of the subscription and not the remaining expiry time. If you then reuse the MQSD in an MQSUB call using the SOALT option you will reset the expiry of the subscription to start counting down again.
SubName	This field is an input field on an MQSUB call and is not changed on output.
SubUserData and SelectionString	<p>These variable length fields will be returned on output from an MQSUB call using the SORES option, if a buffer is provided, and also a positive buffer length in <i>VCHRP</i>. If no buffer is provided only the length will be returned in the <i>VCHRL</i> field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only <i>VCHRP</i> bytes are returned in the provided buffer.</p> <p>If you later reuse the MQSD in an MQSUB call using the SOALT option and a buffer is not provided but a non-zero <i>VCHRL</i> is provided, if that length matches the existing length of the field, no alteration will made to the field.</p>
SubCorrelId and PubAccountingToken	<p>If you do not use SOSCID, then the <i>SDCID</i> will be generated by the queue manager. If you do not use SOSETI, then the <i>SDACC</i> will be generated by the queue manager.</p> <p>These fields will be returned in the MQSD from an MQSUB call using the SORES option. If they are generated by the queue manager, the generated value will be returned on an MQSUB call using the SOCRE or SOALT option.</p>
PubPriority, SubLevel & PubApplIdentityData	These fields will be returned in the MQSD.
ResObjectString	This output only field will be returned in the MQSD if a buffer is provided.

Parameters

The MQSUB call has the following parameters:


HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

SUBDSC (MQSD) - input/output

This is a structure that identifies the object with use that is being registered by the application. See [“MQSD - Subscription descriptor” on page 1169](#) for more information.

HOBJ (10-digit signed integer) - input/output

This handle represents the access that has been established to obtain the messages sent to this subscription. These messages can either be stored on a specific queue or the queue manager can be asked to manage their storage without the need for a specific queue.

Object handle.

If a specific queue is to be used it must be associated with the subscription at creation time. This can be done in two ways:

- By providing this handle when calling MQSUB with the SDCRT option. If this handle is provided as an input parameter on the call, it must be a valid object handle returned from a previous MQOPEN call of a queue using at least one of OOINP*, OOOOUT (if a remote queue for example), or OOBRW option. If this is not the case, the call fails with RC2019. It cannot be an object handle to an alias queue which resolves to a topic object. If so, the call fails with RC2019
- By using the DEFINE SUB MQSC command and provided that command with the name of a queue object.

If the queue manager is to manage the storage of messages sent to this subscription, you should indicate this when the subscription is created, by using the SOMAN option and setting the parameter value to HONONE. The queue manager returns the handle as an output parameter on the call, and the handle that is returned is known as a managed handle. If HONONE is specified and SOMAN is not also specified, the call fails with RC2019.

A managed handle that is returned by the queue manager can be used on an MQGET or MQCB call, with or without browse options, on an MQINQ call, or on MQCLOSE. It cannot be used on MQPUT, MQSET, or on a subsequent MQSUB; attempting to do so fails with RC2039 for MQPUT, RC2040 for MQSET, or RC2038 for MQSUB.

If the SORES option in the *OPTS* field in the MQSD structure is used to resume this subscription, the handle can be returned to the application in this parameter if HONONE is specified. You can use this whether the subscription is using a managed handle or not. It can be useful for subscriptions created using DEFINE SUB if you want the handle to the subscription queue defined on the DEFINE SUB command. In the case where an administratively created subscription is being resumed, the queue is opened with OOINPQ and OOBRW. If other options are needed, the application must open the subscription queue explicitly and provide the object handle on the call. If there is a problem opening the queue the call will fail with RC2522. If the *HOBJ* is provided, it must be equivalent to the *HOBJ* in the original MQSUB call. This means if an object handle returned from an MQOPEN call is being provided, the handle must be to the same queue as previously used or the call fails with RC2019.

If this subscription is being altered, by using the SOALT option in the *OPTS* field in the MQSD structure, then a different *HOBJ* can be provided. Any publications that have been delivered to the queue

previously identified through this parameter remain on that queue and it is the responsibility of the application to retrieve those messages if the *HOBJ* parameter now represents a different queue.

The use of this parameter with various subscription options is summarized in the following table:

Options	Hobj	Description
SOCRT + SOMAN	Ignored on input	Creates a subscription with queue manager managed storage of messages.
SOCRT	Valid object handle	Creates a subscription providing a specific queue as the destination for messages.
SORES	HONONE	Resumes a previously created subscription (managed or not) and have the queue manager return the object handle for use by the application.
SORES	Valid, matching, object handle	Resumes a previously created subscription which uses a specific queue as the destination for messages and use an object handle with specific open options.
SOALT + SOMAN	HONONE	Alters an existing subscription which was previously using a specific queue, to now be managed.
SOALT	Valid object handle	Alters an existing subscription to use a specific queue (either from managed, or from a different specific queue).

Whether it was provided or returned, *HOBJ* must be specified on subsequent MQGET calls that you need to receive the publications.

The *HOBJ* handle ceases to be valid when the MQCLOSE call is issued on it, or when the unit of processing that defines the scope of the handle terminates. The scope of the object handle returned is the same as that of the connection handle specified on the call. See [HCONN](#) for information about handle scope. An MQCLOSE of the *HOBJ* handle has no effect on the *HSUB* handle.

HSUB (10-digit signed integer) - output

This handle represents the subscription that has been made. It can be used for two further operations:

- It can be used on a subsequent MQSUBRQ call to request that publications be sent when the SOPUBR option has been used when making the subscription.
- It can be used on a subsequent MQCLOSE call to remove the subscription that has been made. The *HSUB* handle ceases to be valid when the MQCLOSE call is issued, or when the unit of processing that defines the scope of the handle terminates. The scope of the object handle returned is the same as that of the connection handle specified on the call. An MQCLOSE of the *HSUB* handle has no effect on the *HOBJ* handle.

This handle cannot be passed to an MQGET or MQCB call. You must use the *HOBJ* parameter. Passing this handle to any other IBM MQ call results in RC2019.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion

CCWARN

Warning (partial completion)

CCFAIL

Call failed

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

RC2019

(2019 X'07E3') Object handle not valid

RC2046

(2046 X'07FE') Options not valid or not consistent

RC2085

(2085 X'0825') Object identified cannot be found

RC2161

(2161 X'0871') Queue manager quiescing

RC2298

(2298 X'08FA') Function not supported.

RC2424

(2424 X'0978') Subscription descriptor (MQSD) not valid

RC2425

(2441 X'979') Topic string not valid

RC2428

(2428 X'097C') Subscription name specified does not match existing subscriptions

RC2429

(2429 X'097D') Subscription name exists and is in use by another application

RC2431

(2431 X'097F') SubUserData field not valid

RC2432

(2432 X'0980') Subscription exists

RC2434

(2434 X'0982') Subscription name matches existing subscription

RC2440

(2440 X'0988') SubName field not valid

RC2441

(2441 X'0989') Objectstring field not valid

RC2435

(2435 X'0983') Attribute cannot be changed using SDALT, or subscription was created with SDIMM.

RC2436

(2436 X'0984') SODUR option not valid

RC2459

(2459, X'99B') Selection string syntax error.

RC2503

(2503 X'09C7') MQSUB calls are currently inhibited for the topics subscribed to.

RC2519

(2519, X'9D7') The selection string is not as specified in the description of how to use an MQCHARV structure.

RC2551

(2551, X'9F7') Specified selection string is not available.

RPG Declaration

```

C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQSUB(HCONN : SUBDSC : HOBJ :
C                                HSUB : CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*.1.....2.....3.....4.....5.....6.....7..
DMQSUB          PR          EXTPROC('MQSUB')
D* Connection handle
D HCONN          10I 0 VALUE
D* Subscription descriptor
D SUBDSC          400A
D* Object handle for queue
D HOBJ          10I 0
D* Subscription object handle
D HSUB          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CompCode
D REASON          10I 0

```

MQSUBRQ - Subscription Request

The MQSUBRQ call makes a request on a subscription.

- [“Syntax” on page 1322](#)
- [“Usage notes” on page 1322](#)
- [“Parameters” on page 1323](#)
- [“RPG Declaration” on page 1324](#)

Syntax

MQSUBRQ (*HCONN*, *HSUB*, *ACTION*, *SUBROPT*, *CMPCOD*, *REASON*)

Usage notes

The following usage notes apply to the use of SRAPUB:

1. If this verb completes successfully, the retained publications matching the subscription specified have been sent to the subscription and can be received by using MQGET or MQCB using the HOBJ returned on the original MQSUB verb that created the subscription.
2. If the topic subscribed to by the original MQSUB verb that created the subscription contained a wildcard, more than one retained publication might be sent. The number of publications sent as a result of this call is recorded in the *SRNMP* field in the *SBROPT* structure.
3. If this verb completes with a reason code of RC2437 then there were no currently retained publications for the topic specified.

4. If this verb completes with a reason code of RC2525 or RC2526 then there are currently retained publications for the topic specified but an error has occurred that that meant they were unable to be delivered.
5. The application must have a current subscription to the topic before it can make this call. If the subscription was made in a previous instance of the application and a valid handle to the subscription is not available, the application must first call MQSUB with the SORES option to obtain a handle to it for use in this call.
6. The publications are sent to the destination that is registered for use with the current subscription of this application. If the publications should be sent somewhere else, the subscription must first be altered using the MQSUB call with the SOALT option.

Parameters

The MQSUBRQ call has the following parameters:


HCONN (10-digit signed integer) - input

This handle represents the connection to the queue manager. The value of *HCONN* was returned by a previous MQCONN or MQCONNEX call.

On z/OS for CICS applications, and on IBM i for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN* :

HCDEFH

Default connection handle.

Important:  If you are still using compatibility mode, consider recompiling your application and using static bound calls instead. See [Compatibility mode for RPG applications on IBM i](#)

HSUB (10-digit signed integer) - input

This handle represents the subscription for which an update is to be requested. The value of *HSUB* was returned from a previous MQSUB call.

ACTION (10-digit signed integer) - input

This parameter controls the particular action that is being requested on the subscription. One (and only one) of the following must be specified:

SRAPUB

This action requests that an update publication be sent for the specified topic. This is normally used if the subscriber specified the option SOPUBR on the MQSUB call when it made the subscription. If the queue manager has a retained publication for the topic, this is sent to the subscriber. If not, the call fails. If an application is sent a publication which was retained, this is indicated by the MQIsRetained message property of that publication.

Since the topic in the existing subscription represented by the *HSUB* parameter can contain wildcards, the subscriber might receive multiple retained publications.

SBROPT (MQSRO) - input/output

These options control the action of MQSUBRQ, see [“MQSRO - Subscription request options” on page 562](#) for details.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

CCOK

Successful completion

CCWARN

Warning (partial completion)

CCFAIL

Call failed

Reason (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CPMCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CPMCOD* is CCFAIL:

RC2298

2298 (X'08FA') The function requested is not available in the current environment.

RC2437

2437 (X'0985') There are no retained publications currently stored for this topic.

RC2046

2046 (X'07FE') Options parameter or field contains options that are not valid, or a combination of options that is not valid.

RC2161

2161 (X'0871') Queue manager quiescing

RC2438

2438 (X'0986') On the MQSUBRQ call, the Subscription Request Options MQSRO is not valid.

RPG Declaration

```
C*.1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQSUBRQ(HCONN : HSUB : ACTION :
C                               SBROPT : CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..
DMQSUBRQ      PR          EXTPROC('MQSUBRQ')
D* Connection handle
D HCONN              10I 0 VALUE
D* Subscription handle
D HSUB              10I 0 VALUE
D* Action requested on the subscription
D ACTION            10I 0 VALUE
D* Subscription Request Options
D SBROPT              16A
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CompCode
D REASON              10I 0
```

Attributes of objects

This collection of topics lists only those IBM MQ objects that can be the subject of an MQINQ function call, and gives details of the attributes that can be inquired on and the selectors to be used.

Attributes for queues

Use this information to learn about the different types of queue definitions and the attributes supported by each.

Types of queue: The queue manager supports the following types of queue definition:

Local queue

This is a physical queue that stores messages. The queue exists on the local queue manager.

Applications connected to the local queue manager can place messages on and remove messages from queues of this type. The value of the *QType* queue attribute is QTLOC.

Shared queue

This is a physical queue that stores messages. The queue exists in a shared repository that is accessible to all of the queue managers that belong to the queue-sharing group that owns the shared repository.

Applications connected to any queue manager in the queue-sharing group can place messages on and remove messages from queues of this type. Such queues are effectively the same as local queues. The value of the *QType* queue attribute is QTLOC.

- Shared queues are supported only on z/OS.

Cluster queue

This is a physical queue that stores messages. The queue exists either on the local queue manager, or on one or more of the queue managers that belong to the same cluster as the local queue manager.

Applications connected to the local queue manager can place messages on queues of this type, regardless of the location of the queue. If an instance of the queue exists on the local queue manager, the queue behaves in the same way as a local queue, and applications connected to the local queue manager can remove messages from the queue. The value of the *QType* queue attribute is QTCLUS.

Alias queue

This is not a physical queue - it is an alternative name for a local queue. The name of the local queue to which the alias resolves is part of the definition of the alias queue.

Applications connected to the local queue manager can place messages on and remove messages from alias queues - the messages are placed on and removed from the local queue to which the alias resolves. The value of the *QType* queue attribute is QTALS.

Remote queue

This is not a physical queue - it is the local definition of a queue that exists on a remote queue manager. The local definition of the remote queue contains information that tells the local queue manager how to route messages to the remote queue manager.

Applications connected to the local queue manager can place messages on remote queues - the messages are placed on the local transmission queue used to route messages to the remote queue manager. Applications cannot remove messages from remote queues. The value of the *QType* queue attribute is QTREM.

A remote queue definition can also be used for:

- Reply-queue aliasing

In this case the name of the definition is the name of a reply-to queue. For more information, see [Reply-to queue alias definitions](#).

- Queue-manager aliasing

In this case the name of the definition is an alias for a queue manager, and not the name of a queue. For more information, see [Queue manager alias definitions](#).

Model queue

This is not a physical queue - it is a set of queue attributes from which a local queue can be created.

Messages cannot be stored on queues of this type.

Some queue attributes apply to all types of queue; other queue attributes apply only to certain types of queue. The types of queue to which an attribute applies are indicated by the ✓ symbol in [Table 626 on page 1326](#) and subsequent tables.

[Table 626 on page 1326](#) summarizes the attributes that are specific to queues. The attributes are described in alphabetical order.

Note: The names of the attributes shown in this section are the names used with the MQINQ and MQSET calls. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) Commands](#) for details.

Table 626. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.

Attribute	Description	Local	Model	Alias	Remote	Cluster
AlterationDate	Date when definition was last changed	✓		✓	✓	
AlterationTime	Time when definition was last changed	✓		✓	✓	
BackoutRequeueQName	Excessive backout requeue queue name	✓	✓			
BackoutThreshold	Backout threshold	✓	✓			
BaseQName	Queue name to which alias resolves			✓		
ClusterChannelName	Cluster-sender channel name	✓	✓			
ClusterName	Name of cluster to which queue belongs	✓		✓	✓	
ClusterNamelist	Name of namelist object containing names of clusters to which queue belongs	✓		✓	✓	
CreationDate	Date the queue was created	✓				
CreationTime	Time the queue was created	✓				
CurrentQDepth	Current queue depth	✓				
DefBind	Default binding	✓		✓	✓	✓
DefinitionType	Queue definition type	✓	✓			
DefInputOpenOption	Default input open option	✓	✓			
DefPersistence	Default message persistence	✓	✓	✓	✓	✓
DefPriority	Default message priority	✓	✓	✓	✓	✓
DistLists	Distribution list support	✓	✓			

Table 626. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.

(continued)

Attribute	Description	Local	Model	Alias	Remote	Cluster
HardenGetBackout	Whether to maintain an accurate backout count	✓	✓			
InhibitGet	Controls whether get operations for the queue are allowed	✓	✓	✓		
InhibitPut	Controls whether put operations for the queue are allowed	✓	✓	✓	✓	✓
InitiationQName	Name of initiation queue	✓	✓			
MaxMsgLength	Maximum message length in bytes	✓	✓			
MaxQDepth	Maximum queue depth	✓	✓			
MediaLog	Identity of oldest log extent (or oldest journal receiver on IBM i) needed for media recovery of a specified queue	✓	✓			
MsgDeliverySequence	Message delivery sequence	✓	✓			
OpenInputCount	Number of opens for input	✓				
OpenOutputCount	Number of opens for output	✓				
ProcessName	Process name	✓	✓			
QDepthHighEvent	Controls whether Queue Depth High events are generated	✓	✓			
QDepthHighLimit	High limit for queue depth	✓	✓			
QDepthLowEvent	Controls whether Queue Depth Low events are generated	✓	✓			

Table 626. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.

(continued)

Attribute	Description	Local	Model	Alias	Remote	Cluster
QDepthLowLimit	Low limit for queue depth	✓	✓			
QDepthMaxEvent	Controls whether Queue Full events are generated	✓	✓			
QDesc	Queue description	✓	✓	✓	✓	✓
QName	Queue name	✓		✓	✓	✓
QServiceInterval	Target for queue service interval	✓	✓			
QServiceIntervalEvent	Controls whether Service Interval High or Service Interval OK events are generated	✓	✓			
QType	Queue type	✓		✓	✓	✓
RemoteQMgrName	Name of remote queue manager				✓	
RemoteQName	Name of remote queue				✓	
RetentionInterval	Retention interval	✓	✓			
Scope	Controls whether an entry for the queue also exists in a cell directory	✓		✓	✓	
Shareability	Queue shareability	✓	✓			
TriggerControl	Trigger control	✓	✓			
TriggerData	Trigger data	✓	✓			
TriggerDepth	Trigger depth	✓	✓			
TriggerMsgPriority	Threshold message priority for triggers	✓	✓			
TriggerType	Trigger type	✓	✓			
Usage	Queue usage	✓	✓			

Table 626. Attributes for queues. The columns apply as follows:

- The column for local queues applies also to shared queues.
- The column for model queues indicates which attributes are inherited by the local queue created from the model queue.
- The column for cluster queues indicates the attributes that can be inquired when the cluster queue is opened for inquire alone, or for inquire and output. If the cluster queue is opened for inquire plus one or more of input, browse, or set, the column for local queues applies instead.

(continued)

Attribute	Description	Local	Model	Alias	Remote	Cluster
<u>XmitQName</u>	Transmission queue name				✓	

AlterationDate (12-byte character string)

Date when definition was last changed.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes (for example, 1992-09-23↵↵, where ↵↵ represents two blank characters).

The values of certain attributes (for example, *CurrentQDepth*) change as the queue manager operates. Changes to these attributes do not affect *AlterationDate*.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

This is the time when the definition was last changed. The format of the time is HH.MM.SS using the 24-hour clock, with a leading zero if the hour is less than 10 (for example 09.10.20). The time is local time.

The values of certain attributes (for example, *CurrentQDepth*) change as the queue manager operates. Changes to these attributes do not affect *AlterationTime*.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

BackoutRequeueQName (48-byte character string)

Excessive backout requeue queue name.

Local	Model	Alias	Remote	Cluster
✓	✓			

Applications running inside WebSphere Application Server and those that use the IBM MQ Application Server Facilities use this attribute to determine where messages that have been backed out should go. For all other applications, apart from allowing its value to be queried, the queue manager takes no action based on the value of the attribute.

To determine the value of this attribute, use the CABRQN selector with the MQINQ call. The length of this attribute is given by LNQN.

BackoutThreshold (10-digit signed integer)

Backout threshold.

Local	Model	Alias	Remote	Cluster
✓	✓			

Applications running inside WebSphere Application Server and those that use the IBM MQ Application Server Facilities use this attribute to determine if a message should be backed out. For all other applications, apart from allowing its value to be queried, the queue manager takes no action based on the value of the attribute.

To determine the value of this attribute, use the IABTHR selector with the MQINQ call.

BaseQName (48-byte character string)

The queue name to which the alias resolves.

Local	Model	Alias	Remote	Cluster
		✓		

This is the name of a queue that is defined to the local queue manager. (For more information about queue names, see the description of the *ODON* field in MQOD. The queue is one of the following types:

QTLOC

Local queue.

QTREM

Local definition of a remote queue.

QTCLUS

Cluster queue.

To determine the value of this attribute, use the CABASQ selector with the MQINQ call. The length of this attribute is given by LNQN.

BaseType (integer parameter structure)

The type of object to which the alias resolves.

Local	Model	Alias	Remote	Cluster
		✓		

This attribute can have one of the following values:

OTQ

Base object type is a queue

OTTOP

Base object type is a topic

CFStrucName (12-byte character string)

Coupling-facility structure name.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the name of the coupling-facility structure where the messages on the queue are stored. The first character of the name is in the range A through Z, and the remaining characters are in the range A through Z, 0 through 9, or blank.

The full name of the structure in the coupling facility is obtained by suffixing the value of the *QSGName* queue manager attribute with the value of the *CFStrucName* queue attribute.

This attribute applies only to shared queues; it is ignored if *QSGDisp* does not have the value QSGDSH.

To determine the value of this attribute, use the CACFSN selector with the MQINQ call. The length of this attribute is given by LNCFSN.

This attribute is supported only on z/OS.

ClusterChannelName (20-byte character string)

ClusterChannelName is the generic name of the cluster-sender channels that use this queue as a transmission queue. The attribute specifies which cluster-sender channels send messages to a cluster-receiver channel from this cluster transmission queue.

Local	Model	Alias	Remote	Cluster
✓	✓			

The default queue manager configuration is for all cluster-sender channels to send messages from a single transmission queue, *SYSTEM.CLUSTER.TRANSMIT.QUEUE*. The default configuration can be changed by modified by changing the queue manager attribute, **DefClusterXmitQueueType**. The default value of the attribute is SCTQ. You can change the value to CHANNEL. If you set the **DefClusterXmitQueueType** attribute to CHANNEL, each cluster-sender channel defaults to using a specific cluster transmission queue, *SYSTEM.CLUSTER.TRANSMIT.ChannelName*.

You can also set the transmission queue attribute *ClusterChannelName* attribute to a cluster-sender channel manually. Messages that are destined for the queue manager connected by the cluster-sender channel are stored in the transmission queue that identifies the cluster-sender channel. They are not stored in the default cluster transmission queue. If you set the *ClusterChannelName* attribute to blanks, the channel switches to the default cluster transmission queue when the channel restarts. The default queue is either *SYSTEM.CLUSTER.TRANSMIT.ChannelName* or *SYSTEM.CLUSTER.TRANSMIT.QUEUE*, depending on the value of the queue manager **DefClusterXmitQueueType** attribute.

By specifying asterisks, "*", in **ClusterChannelName**, you can associate a transmission queue with a set of cluster-sender channels. The asterisks can be at the beginning, end, or any number of places in the middle of the channel name string. **ClusterChannelName** is limited to a length of 20 characters: *MQ_CHANNEL_NAME_LENGTH*.

ClusterName (48-byte character string)

Name of cluster to which queue belongs.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

This is the name of the cluster to which the queue belongs. If the queue belongs to more than one cluster, *ClusterNameList* specifies the name of a namelist object that identifies the clusters, and *ClusterName* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank.

To determine the value of this attribute, use the CACLN selector with the MQINQ call. The length of this attribute is given by LNCLUN.

ClusterNameList (48-byte character string)

Name of namelist object containing names of clusters to which queue belongs.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

This is the name of a namelist object that contains the names of clusters to which this queue belongs. If the queue belongs to only one cluster, the namelist object contains only one name. Alternatively, *ClusterName* can be used to specify the name of the cluster, in which case *ClusterNameList* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank.

To determine the value of this attribute, use the CACLNL selector with the MQINQ call. The length of this attribute is given by LNNLN.

CreationDate (12-byte character string)

Date when queue was created.

Local	Model	Alias	Remote	Cluster
✓				

This is the date when the queue was created. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes (for example, 1992-09-23-- , where -- represents two blank characters).

- On IBM i, the creation date of a queue might differ from that of the underlying operating system entity (file or userspace) that represents the queue.

To determine the value of this attribute, use the CACRTD selector with the MQINQ call. The length of this attribute is given by LNCRTD.

CreationTime (8-byte character string)

Time when queue was created.

Local	Model	Alias	Remote	Cluster
✓				

This is the time when the queue was created. The format of the time is HH.MM.SS using the 24-hour clock, with a leading zero if the hour is less than 10 (for example 09.10.20). The time is local time.

- On IBM i, the creation time of a queue might differ from that of the underlying operating system entity (file or user space) that represents the queue.

To determine the value of this attribute, use the CACRTT selector with the MQINQ call. The length of this attribute is given by LNCRTT.

CurrentQDepth (10-digit signed integer)

Current queue depth.

Local	Model	Alias	Remote	Cluster
✓				

This is the number of messages currently on the queue. It is incremented during an MQPUT call, and during backout of an MQGET call. It is decremented during a nonbrowse MQGET call, and during backout of an MQPUT call. The effect of this is that the count includes messages that have been put on the queue within a unit of work, but which have not yet been committed, even though they are not eligible to be retrieved by the MQGET call. Similarly, it excludes messages that have been retrieved within a unit of work using the MQGET call, but which have yet to be committed.

The count also includes messages which have passed their expiry time but have not yet been discarded, although these messages are not eligible to be retrieved. See the *MDEXP* field described in [“MQMD - Message descriptor”](#) on page 1068.

Unit-of-work processing and the segmentation of messages can both cause *CurrentQDepth* to exceed *MaxQDepth*. However, this does not affect the retrievability of the messages - *all* messages on the queue can be retrieved using the MQGET call in the normal way.

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the IACDEP selector with the MQINQ call.

DefBind (10-digit signed integer)

Default binding.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	✓

This attribute is the default binding that is used when OOBNDQ is specified on the MQOPEN call and the queue is a cluster queue. DefBind can have one of the following values:

BNDOPN

Binding fixed by MQOPEN call.

BNDNOT

Binding not fixed.

BNDGRP

Binding is not fixed by the MQOPEN call, but is fixed on MQPUT for all messages in a logical group.

To determine the value of this attribute, use the IADBND selector with the MQINQ call.

DefinitionType (10-digit signed integer)

Queue definition type.

Local	Model	Alias	Remote	Cluster
✓	✓			

This indicates how the queue was defined. The value is one of the following:

QDPRE

Predefined permanent queue.

The queue is a permanent queue created by the system administrator; only the system administrator can delete it.

Predefined queues are created using the DEFINE MQSC command, and can be deleted only by using the DELETE MQSC command. Predefined queues cannot be created from model queues.

Commands can be issued either by an operator, or by an authorized user sending a command message to the command input queue (see the *CommandInputQName* attribute described in [“Attributes for the queue manager”](#) on page 1355).

QDPERM

Dynamically defined permanent queue.

The queue is a permanent queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value QDPERM for the *DefinitionType* attribute.

This type of queue can be deleted using the MQCLOSE call. See [“MQCLOSE - Close object”](#) on page 1223 for more details.

The value of the *QSGDisp* attribute for a permanent dynamic queue is QSGDQM.

QDTEMP

Dynamically defined temporary queue.

The queue is a temporary queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value QDTEMP for the *DefinitionType* attribute.

This type of queue is deleted automatically by the MQCLOSE call when it is closed by the application that created it.

The value of the *QSGDisp* attribute for a temporary dynamic queue is QSGDQM.

QDSHAR

Dynamically defined shared queue.

The queue is a shared permanent queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor MQOD. The model queue definition had the value QDSHAR for the *DefinitionType* attribute.

This type of queue can be deleted using the MQCLOSE call. See [“MQCLOSE - Close object”](#) on page 1223 for more details.

The value of the *QSGDisp* attribute for a shared dynamic queue is QSGDSH.

This attribute in a model queue definition does not indicate how the model queue was defined, because model queues are always predefined. Instead, the value of this attribute in the model queue is used to determine the *DefinitionType* of each of the dynamic queues created from the model queue definition using the MQOPEN call.

To determine the value of this attribute, use the IADEFT selector with the MQINQ call.

DefInputOpenOption (10-digit signed integer)

Default input open option.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the default way in which the queue should be opened for input. It applies if the OOINPQ option is specified on the MQOPEN call when the queue is opened. This can have one of the following values:

OOINPX

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code RC2042 if the queue is currently open by this or another application for input of any type (OOINPS or OOIINPX).

OOINPS

Open queue to get messages with shared access.

The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with OOIINPS, but fails with reason code RC2042 if the queue is currently open with OOIINPX.

To determine the value of this attribute, use the IADINP selector with the MQINQ call.

DefPersistence (10-digit signed integer)

Default message persistence.

Local	Model	Alias	Remote	Cluster
✓	✓	✓	✓	✓

This is the default persistence of messages on the queue. It applies if PEQDEF is specified in the message descriptor when the message is put.

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path at the time of the MQPUT or MQPUT1 call. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

This can have one of the following values:

PEPER

Message is persistent.

This means that the message survives system failures and restarts of the queue manager. Persistent messages cannot be placed on:

- Temporary dynamic queues
- Shared queues

Persistent messages can be placed on permanent dynamic queues, and predefined queues.

PENPER

Message is not persistent.

This means that the message does not normally survive system failures or restarts of the queue manager. This applies even if an intact copy of the message is found on auxiliary storage during restart of the queue manager.

In the special case of shared queues, nonpersistent messages *do* survive restarts of queue managers in the queue-sharing group, but do not survive failures of the coupling facility used to store messages on the shared queues.

Both persistent and nonpersistent messages can exist on the same queue.

To determine the value of this attribute, use the IADPER selector with the MQINQ call.

DefPriority (10-digit signed integer)

Default message priority.

Local	Model	Alias	Remote	Cluster
✓	✓	✓	✓	✓

This is the default priority for messages on the queue. This applies if PRQDEF is specified in the message descriptor when the message is put on the queue.

If there is more than one definition in the queue-name resolution path, the default priority for the message is taken from the value of this attribute in the *first* definition in the path at the time of the put operation. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The way in which a message is placed on a queue depends on the value of the queue's *MsgDeliverySequence* attribute:

- If the *MsgDeliverySequence* attribute is MSPRIO, the logical position at which a message is placed on the queue is dependent on the value of the *MDPRI* field in the message descriptor.
- If the *MsgDeliverySequence* attribute is MSFIFO, messages are placed on the queue as though they had a priority equal to the *DefPriority* of the resolved queue, regardless of the value of the *MDPRI* field in the message descriptor. However, the *MDPRI* field retains the value specified by the application that put the message. See the *MsgDeliverySequence* attribute described in [“Attributes for queues”](#) on page 1324 for more information.

Priorities are in the range zero (lowest) through *MaxPriority* (highest); see the *MaxPriority* attribute described in [“Attributes for the queue manager”](#) on page 1355.

To determine the value of this attribute, use the IADPRI selector with the MQINQ call.

DefReadAhead (10-digit signed integer)

Specifies the default read ahead behavior for non-persistent messages delivered to the client.

Local	Model	Alias	Remote	Cluster
✓	✓	✓		

DefReadAhead can be set to one of the following values:

RAHNO

Non-persistent messages are not sent ahead to the client before an application requests them. A maximum of one non-persistent message can be lost if the client ends abnormally.

RAHYES

Non-persistent messages are sent ahead to the client before an application requests them. Non-persistent messages can be lost if the client ends abnormally or if the client does not consume all the messages it is sent.

RAHDIS

Read ahead of non-persistent messages is not enabled for this queue. Messages are not sent ahead to the client regardless of whether read ahead is requested by the client application.

To determine the value of this attribute, use the IADRAH selector with the MQINQ call.

DefPResp (10-digit signed integer)

The default put response type (DEFPRESP) attribute defines the value used by applications when the PutResponseType within MQPMO has been set to PMRASQ. This attribute is valid for all queue types.

Local	Model	Alias	Remote	Cluster
✓	✓	✓	✓	✓

This can have one of the following values:

SYNC

The put operation is issued synchronously returning a response.

ASYNC

The put operation is issued asynchronously, returning a subset of MQMD fields.

To determine the value of this attribute, use the IADPRT selector with the MQINQ call.

DistLists (10-digit signed integer)

Distribution list support.

Local	Model	Alias	Remote	Cluster
✓	✓			

This indicates whether distribution-list messages can be placed on the queue. The attribute is set by a message channel agent (MCA) to inform the local queue manager whether the queue manager at the other end of the channel supports distribution lists. This latter queue manager (called the "partnering queue manager") is the one which next receives the message, after it has been removed from the local transmission queue by a sending MCA.

The attribute is set by the sending MCA whenever it establishes a connection to the receiving MCA on the partnering queue manager. In this way, the sending MCA can cause the local queue manager to place on the transmission queue only messages which the partnering queue manager can process correctly.

This attribute is primarily for use with transmission queues, but the processing described is performed regardless of the usage defined for the queue (see the *Usage* attribute).

This can have one of the following values:

DLSUPP

Distribution lists supported.

This indicates that distribution-list messages can be stored on the queue, and transmitted to the partnering queue manager in that form. This reduces the amount of processing required to send the message to multiple destinations.

DLNSUP

Distribution lists not supported.

This indicates that distribution-list messages cannot be stored on the queue, because the partnering queue manager does not support distribution lists. If an application puts a distribution-list message, and that message is to be placed on this queue, the queue manager splits the distribution-list message and places the individual messages on the queue instead. This increases the amount of processing required to send the message to multiple destinations, but ensures that the messages will be processed correctly by the partnering queue manager.

To determine the value of this attribute, use the IADIST selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

HardenGetBackout (10-digit signed integer)

Whether to maintain an accurate backout count.

Local	Model	Alias	Remote	Cluster
✓	✓			

For each message, a count is kept of the number of times that the message is retrieved by an MQGET call within a unit of work, and that unit of work later backed out. This count is available in the *MDBOC* field in the message descriptor after the MQGET call has completed.

The message backout count survives when the queue manager restarts. However, to ensure that the count is accurate, information has to be "hardened" (recorded on disk or other permanent storage device) each time a message is retrieved by an MQGET call within a unit of work for this queue. If this is not done, and a failure of the queue manager occurs together with backout of the MQGET call, the count might not be incremented.

Hardening information for each MQGET call within a unit of work, however, imposes a performance cost, and the *HardenGetBackout* attribute should be set to QABH only if the count has to be accurate.

- On IBM i, the message backout count is always hardened, regardless of the setting of this attribute.

The following values are possible:

QABH

Backout count remembered.

Hardening is used to ensure that the backout count for messages on this queue is accurate.

QABNH

Backout count might not be remembered.

Hardening is not used to ensure that the backout count for messages on this queue is accurate. The count might therefore be lower than it should be.

To determine the value of this attribute, use the IAHGB selector with the MQINQ call.

InhibitGet (10-digit signed integer)

Controls whether get operations for this queue are allowed.

Local	Model	Alias	Remote	Cluster
✓	✓	✓		

If the queue is an alias queue, get operations must be allowed for both the alias and the base queue at the time of the get operation, in order for the MQGET call to succeed. The value is one of the following:

QAGETI

Get operations are inhibited.

MQGET calls fail with reason code RC2016. This includes MQGET calls that specify GMBRWF or GMBRWN.

Note: If an MQGET call operating within a unit of work completes successfully, changing the value of the *InhibitGet* attribute after to QAGETI does not prevent the unit of work being committed.

QAGETA

Get operations are allowed.

To determine the value of this attribute, use the IAIGET selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InhibitPut (10-digit signed integer)

Controls whether put operations for this queue are allowed.

Local	Model	Alias	Remote	Cluster
✓	✓	✓	✓	✓

If there is more than one definition in the queue-name resolution path, put operations must be allowed for *every* definition in the path (including any queue manager alias definitions) at the time of the put operation, in order for the MQPUT or MQPUT1 call to succeed. This can have one of the following values:

QAPUTI

Put operations are inhibited.

MQPUT and MQPUT1 calls fail with reason code RC2051.

Note: If an MQPUT call operating within a unit of work completes successfully, changing the value of the *InhibitPut* attribute later to QAPUTI does not prevent the unit of work being committed.

QAPUTA

Put operations are allowed.

To determine the value of this attribute, use the IAIPUT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InitiationQName (48-byte character string)

Name of initiation queue.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the name of a queue defined on the local queue manager; the queue must be of type QTLOC. The queue manager sends a trigger message to the initiation queue when application startup is required as a result of a message arriving on the queue to which this attribute belongs. The initiation queue must be monitored by a trigger monitor application which will start the appropriate application after receipt of the trigger message.

To determine the value of this attribute, use the CAINIQ selector with the MQINQ call. The length of this attribute is given by LNQN.

MaxMsgLength (10-digit signed integer)

Maximum message length in bytes.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is an upper limit for the length of the longest *physical* message that can be placed on the queue. However, because the *MaxMsgLength* queue attribute can be set independently of the *MaxMsgLength* queue manager attribute, the actual upper limit for the length of the longest physical message that can be placed on the queue is the lesser of those two values.

If the queue manager supports segmentation, it is possible for an application to put a *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MFSEGA flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but typically, resource constraints imposed by the operating system or by the environment in which the application is running, results in a lower limit.

An attempt to place on the queue a message that is too long fails with reason code:

- RC2030 if the message is too large for the queue
- RC2031 if the message is too large for the queue manager, but not too large for the queue

The lower limit for the *MaxMsgLength* attribute is zero. The upper limit is determined by the environment:

- On IBM i, the maximum message length is 100 MB (104 857 600 bytes).

For more information, see the *BUFLen* parameter described in “MQPUT - Put message” on page 1289.

To determine the value of this attribute, use the IAMLEN selector with the MQINQ call.

MaxQDepth (10-digit signed integer)

Maximum queue depth.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the defined upper limit for the number of physical messages that can exist on the queue at any one time. An attempt to put a message on a queue that already contains *MaxQDepth* messages, fails with reason code RC2053.

Unit-of-work processing and the segmentation of messages can both cause the actual number of physical messages on the queue to exceed *MaxQDepth*. However, this does not affect the retrievability of the messages - *all* messages on the queue can be retrieved using the MQGET call in the normal way.

The value of this attribute is zero or greater. The upper limit is determined by the environment.

Note: It is possible for the storage space available to the queue to be exhausted even if there are fewer than *MaxQDepth* messages on the queue.

To determine the value of this attribute, use the IAMDEP selector with the MQINQ call.

MediaLog (10-digit signed integer)

Identity of the log extent (or journal receiver on IBM i) needed for media recovery of a particular queue.

Local	Model	Alias	Remote	Cluster
✓	✓			

On queue managers where circular logging is in use, the value is returned as a null string.

MsgDeliverySequence (10-digit signed integer)

Message delivery sequence.

Local	Model	Alias	Remote	Cluster
✓	✓			

This determines the order in which messages are returned to the application by the MQGET call:

MSFIFO

Messages are returned in FIFO order (first in, first out).

This means that an MQGET call will return the *first* message that satisfies the selection criteria specified on the call, regardless of the priority of the message.

MSPRIO

Messages are returned in priority order.

This means that an MQGET call will return the *highest-priority* message that satisfies the selection criteria specified on the call. Within each priority level, messages are returned in FIFO order (first in, first out).

If the relevant attributes are changed while there are messages on the queue, the delivery sequence is as follows:

- The order in which messages are returned by the MQGET call is determined by the values of the *MsgDeliverySequence* and *DefPriority* attributes in force for the queue at the time the message arrives on the queue:
 - If *MsgDeliverySequence* is MSFIFO when the message arrives, the message is placed on the queue as though its priority were *DefPriority*. This does not affect the value of the *MDPRI* field in the message descriptor of the message; that field retains the value it had when the message was first put.
 - If *MsgDeliverySequence* is MSPRIO when the message arrives, the message is placed on the queue at the place appropriate to the priority given by the *MDPRI* field in the message descriptor.

If the value of the *MsgDeliverySequence* attribute is changed while there are messages on the queue, the order of the messages on the queue is not changed.

If the value of the *DefPriority* attribute is changed while there are messages on the queue, the messages will not necessarily be delivered in FIFO order, even though the *MsgDeliverySequence* attribute is set to MSFIFO; those that were placed on the queue at the higher priority are delivered first.

To determine the value of this attribute, use the IAMDS selector with the MQINQ call.

OpenInputCount (10-digit signed integer)

Number of opens for input.

Local	Model	Alias	Remote	Cluster
✓				

This is the number of handles that are currently valid for removing messages from the queue with the MQGET call. It is the total number of such handles known to the *local* queue manager. If the queue is a shared queue, the count does not include opens for input that were performed for the queue at other queue managers in the queue-sharing group to which the local queue manager belongs.

The count includes handles where an alias queue which resolves to this queue was opened for input. The count does not include handles where the queue was opened for actions which did not include input (for example, a queue opened only for browse).

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the IAOIC selector with the MQINQ call.

OpenOutputCount (10-digit signed integer)

Number of opens for output.

Local	Model	Alias	Remote	Cluster
✓				

This is the number of handles that are currently valid for adding messages to the queue with the MQPUT call. It is the total number of such handles known to the *local* queue manager; it does not include opens for output that were performed for this queue at remote queue managers. If the queue is a shared queue, the count does not include opens for output that were performed for the queue at other queue managers in the queue-sharing group to which the local queue manager belongs.

The count includes handles where an alias queue which resolves to this queue was opened for output. The count does not include handles where the queue was opened for actions which did not include output (for example, a queue opened only for inquire).

The value of this attribute fluctuates as the queue manager operates.

To determine the value of this attribute, use the IAOOC selector with the MQINQ call.

ProcessName (48-byte character string)

Process name.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the name of a process object that is defined on the local queue manager. The process object identifies a program that can service the queue.

To determine the value of this attribute, use the CAPRON selector with the MQINQ call. The length of this attribute is given by LNPRON.

QDepthHighEvent (10-digit signed integer)

Controls whether Queue Depth High events are generated.

Local	Model	Alias	Remote	Cluster
✓	✓			

A Queue Depth High event indicates that an application has put a message on a queue, which has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold (see the *QDepthHighLimit* attribute).

Note: The value of this attribute can change dynamically.

QDepthHighEvent can have one of two values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAQDHE selector with the MQINQ call.

QDepthHighLimit (10-digit signed integer)

High limit for queue depth.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the threshold against which the queue depth is compared to generate a Queue Depth High event. This event indicates that an application has put a message on a queue, and this has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold. See the *QDepthHighEvent* attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is in the range zero through 100. The default value is 80.

To determine the value of this attribute, use the IAQDHL selector with the MQINQ call.

QDepthLowEvent (10-digit signed integer)

Controls whether Queue Depth Low events are generated.

Local	Model	Alias	Remote	Cluster
✓	✓			

A Queue Depth Low event indicates that an application has retrieved a message from a queue, which has caused the number of messages on the queue to become less than or equal to the queue depth low threshold (see the *QDepthLowLimit* attribute).

Note: The value of this attribute can change dynamically.

QDepthLowEvent can have one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAQDLE selector with the MQINQ call.

QDepthLowLimit (10-digit signed integer)

Low limit for queue depth.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the threshold against which the queue depth is compared to generate a Queue Depth Low event. This event indicates that an application has retrieved a message from a queue, and this has caused the number of messages on the queue to become less than or equal to the queue depth low threshold. See the *QDepthLowEvent* attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is in the range zero through 100. The default value is 20.

To determine the value of this attribute, use the IAQDLL selector with the MQINQ call.

QDepthMaxEvent (10-digit signed integer)

Controls whether Queue Full events are generated.

Local	Model	Alias	Remote	Cluster
✓	✓			

A Queue Full event indicates that a put to a queue has been rejected because the queue is full, that is, the queue depth has already reached its maximum value.

Note: The value of this attribute can change dynamically.

This can have one of the following values:

EVARDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAQDME selector with the MQINQ call.

QDesc (64-byte character string)

Queue description.

Local	Model	Alias	Remote	Cluster
✓	✓	✓	✓	✓

This is a field that can be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contains only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the CAQD selector with the MQINQ call. The length of this attribute is given by LNQD.

QName (48-byte character string)

Queue name.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	✓

This is the name of a queue defined on the local queue manager. For more information about queue names, see [Rules for naming IBM MQ objects](#). All queues defined on a queue manager share the same queue namespace. Therefore, a QTLOC queue and a QTALS queue cannot have the same name.

To determine the value of this attribute, use the CAQN selector with the MQINQ call. The length of this attribute is given by LNQN.

QServiceInterval (10-digit signed integer)

Target for queue service interval.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the service interval used for comparison to generate Service Interval High and Service Interval OK events. See the *QServiceIntervalEvent* attribute.

The value is in units of milliseconds, and is in the range zero through 999 999 999.

To determine the value of this attribute, use the IAQSI selector with the MQINQ call.

QServiceIntervalEvent (10-digit signed integer)

Controls whether Service Interval High or Service Interval OK events are generated.

Local	Model	Alias	Remote	Cluster
✓	✓			

- A Service Interval High event is generated when a check indicates that no messages have been retrieved from the queue for at least the time indicated by the *QServiceInterval* attribute.
- A Service Interval OK event is generated when a check indicates that messages have been retrieved from the queue within the time indicated by the *QServiceInterval* attribute.

Note: The value of this attribute can change dynamically.

This attribute can have one of the following values:

QSIEHI

Queue Service Interval High events enabled.

- Queue Service Interval High events are **enabled** and
- Queue Service Interval OK events are **disabled**.

QSIEOK

Queue Service Interval OK events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are **enabled**.

QSIENO

No queue service interval events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are also **disabled**.

For shared queues, the value of this attribute is ignored; the value QSIENO is assumed.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAQSIE selector with the MQINQ call.

QSGDisp (10-digit signed integer)

Queue-sharing group disposition.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

This specifies the disposition of the queue. The value is one of the following:

QSGDQM

Queue manager disposition.

The object has queue manager disposition. This means that the object definition is known only to the local queue manager; the definition is not known to other queue managers in the queue-sharing group.

It is possible for each queue manager in the queue-sharing group to have an object with the same name and type as the current object, but these are separate objects and there is no correlation between them. Their attributes are not constrained to be the same as each other.

QSGDCP

Copied-object disposition.

The object is a local copy of a master object definition that exists in the shared repository. Each queue manager in the queue-sharing group can have its own copy of the object. Initially, all copies have the same attributes, but by using MQSC commands each copy can be altered so that its attributes differ from those of the other copies. The attributes of the copies are resynchronized when the master definition in the shared repository is altered.

QSGDSH

Shared disposition.

The object has shared disposition. This means that there exists in the shared repository a single instance of the object that is known to all queue managers in the queue-sharing group. When a queue manager in the group accesses the object, it accesses the single shared instance of the object.

To determine the value of this attribute, use the IAQSGD selector with the MQINQ call.

This attribute is supported only on z/OS.

QType (10-digit signed integer)

Queue type.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	✓

This attribute has one of the following values:

QTALS

Alias queue definition.

QTCLUS

Cluster queue.

QTLOC

Local queue.

QTREM

Local definition of a remote queue.

To determine the value of this attribute, use the IAQTYP selector with the MQINQ call.

RemoteQMgrName (48-byte character string)

Name of remote queue manager.

Local	Model	Alias	Remote	Cluster
			✓	

This is the name of the remote queue manager on which the queue *RemoteQName* is defined. If the *RemoteQName* queue has a *QSGDisp* value of *QSGDCP* or *QSGDSH*, *RemoteQMgrName* can be the name of the queue-sharing group that owns *RemoteQName*.

If an application opens the local definition of a remote queue, *RemoteQMgrName* must not be blank and must not be the name of the local queue manager. If *XmitQName* is blank, the local queue with same name as *RemoteQMgrName* is used as the transmission queue. If there is no queue with the name *RemoteQMgrName*, the queue identified by the *DefXmitQName* queue manager attribute is used.

If this definition is used for a queue manager alias, *RemoteQMgrName* is the name of the queue manager that is being aliased. It can be the name of the local queue manager. Otherwise, if *XmitQName* is blank when the open occurs, there must be a local queue with the same name as *RemoteQMgrName*; this queue is used as the transmission queue.

If this definition is used for a reply-to alias, this name is the name of the queue manager which is to be the *MDRM*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the *CARQMN* selector with the *MQINQ* call. The length of this attribute is given by *LNQMN*.

RemoteQName (48-byte character string)

Name of remote queue.

Local	Model	Alias	Remote	Cluster
			✓	

This is the name of the queue as it is known on the remote queue manager *RemoteQMgrName*.

If an application opens the local definition of a remote queue, when the open occurs *RemoteQName* must not be blank.

If this definition is used for a queue manager alias definition, when the open occurs *RemoteQName* must be blank.

If the definition is used for a reply-to alias, this name is the name of the queue that is to be the *MDRQ*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the *CARQN* selector with the *MQINQ* call. The length of this attribute is given by *LNQN*.

RetentionInterval (10-digit signed integer)

Retention interval.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the time which the queue should be retained. After this time has elapsed, the queue is eligible for deletion.

The time is measured in hours, counting from the date and time when the queue was created. The creation date of the queue is recorded in the *CreationDate* and the create time of the queue is recorded in the *CreationTime* attribute.

This information is provided to enable a housekeeping application or the operator to identify and delete queues that are no longer required.

Note: The queue manager never tries to delete queues based on this attribute, or to prevent the deletion of queues with a retention interval that has not expired; it is the user's responsibility to cause any required action to be taken.

A realistic retention interval should be used to prevent the accumulation of permanent dynamic queues (see *DefinitionType*). However, this attribute can also be used with predefined queues.

To determine the value of this attribute, use the IARINT selector with the MQINQ call.

Scope (10-digit signed integer)

Controls whether an entry for this queue also exists in a cell directory.

Local	Model	Alias	Remote	Cluster
✓		✓	✓	

A cell directory is provided by an installable Name service. This can have one of the following values:

SCOQM

Queue-manager scope.

The queue definition has queue manager scope. This means that the definition of the queue does not extend beyond the queue manager which owns it. To open the queue for output from some other queue manager, either the name of the owning queue manager must be specified, or the other queue manager must have a local definition of the queue.

SCOCEL

Cell scope.

The queue definition has cell scope. This means that the queue definition is also placed in a cell directory available to all of the queue managers in the cell. The queue can be opened for output from any of the queue managers in the cell merely by specifying the name of the queue; the name of the queue manager which owns the queue need not be specified. However, the queue definition is not available to any queue manager in the cell which also has a local definition of a queue with that name, as the local definition takes precedence.

A cell directory is provided by an installable name service such as LDAP (Lightweight Directory Access Protocol). Note that IBM MQ no longer supports the DCE (Distributed Computing Environment) name service that was formerly used for inserting queue definitions into a DCE directory (also no longer supported).

Model and dynamic queues cannot have cell scope.

This value is only valid if a name service supporting a cell directory has been configured.

To determine the value of this attribute, use the IASCP selector with the MQINQ call.

Support for this attribute is subject to the following restrictions:

- On IBM i, the attribute is supported, but only SCOQM is valid.

Shareability (10-digit signed integer)

Whether queue can be shared for input.

Local	Model	Alias	Remote	Cluster
✓	✓			

This indicates whether the queue can be opened for input multiple times concurrently. This can have one of the following values:

QASHR

Queue is shareable.

Multiple opens with the OOINPS option are allowed.

QANSHR

Queue is not shareable.

An MQOPEN call with the OOINPS option is treated as OOINPX.

To determine the value of this attribute, use the IASHAR selector with the MQINQ call.

TriggerControl (10-digit signed integer)

Trigger control.

Local	Model	Alias	Remote	Cluster
✓	✓			

This controls whether trigger messages are written to an initiation queue, in order to cause an application to be started to service the queue. This is one of the following:

TCOFF

Trigger messages not required.

No trigger messages are to be written for this queue. The value of *TriggerType* is irrelevant in this case.

TCON

Trigger messages required.

Trigger messages are to be written for this queue, when the appropriate trigger events occur.

To determine the value of this attribute, use the IATRGC selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerData (64-byte character string)

Trigger data.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is free-format data that the queue manager inserts into the trigger message when a message arriving on this queue causes a trigger message to be written to the initiation queue.

The content of this data is of no significance to the queue manager. It is meaningful either to the trigger-monitor application which processes the initiation queue, or to the application which is started by the trigger monitor.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the CATRGD selector with the MQINQ call. To change the value of this attribute, use the MQSET call. The length of this attribute is given by LNTRGD.

TriggerDepth (10-digit signed integer)

Trigger depth.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the number of messages of priority *TriggerMsgPriority* or greater that must be on the queue before a trigger message is written. This applies when *TriggerType* is set to TTDPTH. The value of *TriggerDepth* is one or greater. This attribute is not used otherwise.

To determine the value of this attribute, use the IATRGD selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerMsgPriority (10-digit signed integer)

Threshold message priority for triggers.

Local	Model	Alias	Remote	Cluster
✓	✓			

This is the message priority below which messages do not contribute to the generation of trigger messages (that is, the queue manager ignores these messages when determining whether a trigger message should be generated). *TriggerMsgPriority* can be in the range zero (lowest) through *MaxPriority* (highest; see [“Attributes for the queue manager”](#) on page 1355); a value of zero causes all messages to contribute to the generation of trigger messages.

To determine the value of this attribute, use the IATRGP selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerType (10-digit signed integer)

Trigger type.

Local	Model	Alias	Remote	Cluster
✓	✓			

This controls the conditions under which trigger messages are written as a result of messages arriving on this queue. The value is one of the following:

TTNONE

No trigger messages.

No trigger messages are written as a result of messages on this queue. This has the same effect as setting *TriggerControl* to TCOFF.

TTFIRST

Trigger message when queue depth goes from 0 to 1.

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue changes from 0 to 1.

TTEVRY

Trigger message for every message.

A trigger message is written whenever a message of priority *TriggerMsgPriority* or greater arrives on the queue.

TTDPTH

Trigger message when depth threshold exceeded.

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue equals or exceeds *TriggerDepth*. After the trigger message has been written, *TriggerControl* is set to TCOFF to prevent further triggering until it is explicitly turned on again.

To determine the value of this attribute, use the IATRGT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

Usage (10-digit signed integer)

Queue usage.

Local	Model	Alias	Remote	Cluster
✓	✓			

This indicates what the queue is used for. The value is one of the following:

USNORM

Normal usage.

This is a queue that normal applications use when putting and getting messages; the queue is not a transmission queue.

USTRAN

Transmission queue.

This is a queue used to hold messages destined for remote queue managers. When a normal application sends a message to a remote queue, the local queue manager stores the message temporarily on the appropriate transmission queue in a special format. A message channel agent then reads the message from the transmission queue, and transports the message to the remote queue manager. For more information about transmission queues, see [Transmission queues](#).

Only privileged applications can open a transmission queue for OOOOUT to put messages on it directly. Only utility applications would normally be expected to do this. Care must be taken that the message data format is correct (see “MQXQH - Transmission-queue header” on page 1201), otherwise errors might occur during the transmission process. Context is not passed or set unless one of the PM* context options is specified.

To determine the value of this attribute, use the IAUSAG selector with the MQINQ call.

XmitQName (48-byte character string)

Transmission queue name.

Local	Model	Alias	Remote	Cluster
			✓	

If this attribute is nonblank when an open occurs, either for a remote queue or for a queue manager alias definition, it specifies the name of the local transmission queue to be used for forwarding the message.

If *XmitQName* is blank, the local queue with the same name as *RemoteQMGrName* is used as the transmission queue. If there is no queue with the name *RemoteQMGrName*, the queue identified by the *DefXmitQName* queue manager attribute is used.

This attribute is ignored if the definition is being used as a queue manager alias and *RemoteQMGrName* is the name of the local queue manager. It is also ignored if the definition is used as a reply-to queue alias definition.

To determine the value of this attribute, use the CAXQN selector with the MQINQ call. The length of this attribute is given by LNQN.

Attributes for namelists

This topic summarizes the attributes that are specific to namelists. The attributes are described in alphabetical order.

Note: The names of the attributes shown are the names used with the MQINQ and MQSET calls.

Attribute descriptions

A namelist object has the following attributes:

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

NameCount (10-digit signed integer)

Number of names in namelist.

This is greater than or equal to zero. The following value is defined:

NCMXNL

Maximum number of names in a namelist.

To determine the value of this attribute, use the IANAMC selector with the MQINQ call.

NamelistDesc (64-byte character string)

Namelist description.

This is a field that might be used for descriptive commentary; its value is established by the definition process. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contains only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the CALSTD selector with the MQINQ call.

The length of this attribute is given by LNNLD.

NamelistName (48-byte character string)

Namelist name.

This is the name of a namelist that is defined on the local queue manager.

Each namelist has a name that is different from the names of other namelists belonging to the queue manager, but might duplicate the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the CALSTN selector with the MQINQ call.

The length of this attribute is given by LNNLN.

Names (48-byte character string x NameCount)

A list of *NameCount* names.

Each name is the name of an object that is defined to the local queue manager. For more information about object names, see [Naming IBM MQ objects](#).

To determine the value of this attribute, use the CANAMS selector with the MQINQ call.

The length of each name in the list is given by LNOBJN.

Attributes for process definitions

This topic summarizes the attributes that are specific to process definitions. The attributes are described in alphabetical order.

Note: The names of the attributes shown are the names used with the MQINQ and MQSET calls. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [MQSC commands](#) for details.

Attribute descriptions

A process-definition object has the following attributes:

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

ApplId (256-byte character string)

Application identifier.

This is a character string that identifies the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

The meaning of *ApplId* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ requires *ApplId* to be the name of an executable program.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the CAAPPI selector with the MQINQ call. The length of this attribute is given by LNPROA.

ApplType (10-digit signed integer)

Application type.

This identifies the nature of the program to be started in response to the receipt of a trigger message. This information is for use by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

ApplType can have any value. You can use the following values for standard types; user-defined application types are restricted to values in the range ATUFST through ATULST:

ATCICS

CICS transaction.

AT400

IBM i application.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

To determine the value of this attribute, use the IAAPPT selector with the MQINQ call.

EnvData (128-byte character string)

Environment data.

This is a character string that contains environment-related information pertaining to the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

The meaning of *EnvData* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ appends *EnvData* to the parameter list passed to the started application. The parameter list consists of the MQTMC2 structure, followed by one blank, followed by *EnvData* with trailing blanks removed.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the CAENVD selector with the MQINQ call. The length of this attribute is given by LNPROE.

ProcessDesc (64-byte character string)

Process description.

This is a field that can be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the CAPROD selector with the MQINQ call.

The length of this attribute is given by LNPROD.

ProcessName (48-byte character string)

Process name.

This is the name of a process definition that is defined on the local queue manager.

Each process definition has a name that is different from the names of other process definitions belonging to the queue manager. But the name of the process definition can be the same as the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the CAPRON selector with the MQINQ call.

The length of this attribute is given by LNPRON.

UserData (128-byte character string)

User data.

This is a character string that contains user information pertaining to the application to be started. This information is for use by a trigger-monitor application that processes messages on the initiation queue, or the application which is started by the trigger monitor. The information is sent to the initiation queue as part of the trigger message.

The meaning of *UserData* is determined by the trigger-monitor application. The trigger monitor provided by IBM MQ passes *UserData* to the started application as part of the parameter list. The parameter list consists of the MQTMC2 structure (containing *UserData*), followed by one blank, followed by *EnvData* with trailing blanks removed.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the CAUSRD selector with the MQINQ call. The length of this attribute is given by LNPROU.

Attributes for the queue manager

A summary of queue manager attributes.

Some queue manager attributes are fixed for particular implementations, while others can be changed by using the MQSC command ALTER QMGR. The attributes can also be displayed by using the command DISPLAY QMGR. Most queue manager attributes can be inquired by opening a special OTQM object, and using the MQINQ call with the handle returned.

The following table summarizes the attributes that are specific to the queue manager. The attributes are described in alphabetical order.

Note: The names of the attributes shown in this section are the names used with the MQINQ and MQSET calls. When MQSC commands are used to define, alter, or display attributes, alternative short names are used; see [Script \(MQSC\) Commands](#) for more information.

Table 627. Attributes for the queue manager	
Attribute	Description
AlterationDate	Date when definition was last changed
AlterationTime	Time when definition was last changed
AuthorityEvent	Controls whether authorization (Not Authorized) events are generated
BridgeEvent	Controls whether IMS bridge events are generated
ChannelAutoDef	Controls whether automatic channel definition is permitted
ChannelAutoDefEvent	Controls whether channel automatic-definition events are generated
ChannelAutoDefExit	Name of user exit for automatic channel definition
ChannelEvent	Controls whether channel events are generated
ClusterCacheType	Controls whether the cluster cache is fixed in size or dynamically sized
ClusterWorkloadData	User data for cluster workload exit

<i>Table 627. Attributes for the queue manager (continued)</i>	
Attribute	Description
<u>ClusterWorkloadExit</u>	Name of user exit for cluster workload management
<u>ClusterWorkloadLength</u>	Maximum length of message data passed to cluster workload exit
<u>CodedCharSetId</u>	Coded character set identifier
<u>CommandEvent</u>	Controls whether command event messages are queued
<u>CommandInputQName</u>	Command input queue name
<u>CommandLevel</u>	Command level
<u>ConfigurationEvent</u>	Configuration event
<u>DeadLetterQName</u>	Name of dead-letter queue
<u>DefClusterXmitQueueType</u>	Default cluster transmission queue type
<u>DefXmitQName</u>	Default transmission queue name
<u>DistLists</u>	Distribution list support
<u>InhibitEvent</u>	Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated
<u>LocalEvent</u>	Controls whether local error events are generated
<u>LoggerEvent</u>	Controls whether recovery log events are generated
<u>MaxHandles</u>	Maximum number of handles
<u>MaxMsgLength</u>	Maximum message length in bytes
<u>MaxPriority</u>	Maximum priority
<u>MaxUncommittedMsgs</u>	Maximum number of uncommitted messages within a unit of work
<u>PerformanceEvent</u>	Controls whether performance-related events are generated
<u>Platform</u>	Platform on which the queue manager is running
<u>PubSubMode</u>	Whether the publish/subscribe engine and queued publish/subscribe interface are running
<u>QMGrDesc</u>	Queue manager description
<u>QMGrIdentifier</u>	Unique internally-generated identifier of queue manager
<u>QMGrName</u>	Queue manager name
<u>RemoteEvent</u>	Controls whether remote error events are generated
<u>RepositoryName</u>	Name of cluster for which this queue manager provides repository services
<u>RepositoryNamelist</u>	Name of namelist object containing names of clusters for which this queue manager provides repository services
<u>SSLCRLNamelist</u>	Name of namelist object containing names of authentication information objects (See Note 1)
<u>SSLEvent</u>	Controls whether SSL events are generated
<u>SSLKeyRepository</u>	Location of SSL key repository (See Note 1)

Table 627. Attributes for the queue manager (continued)	
Attribute	Description
SSLKeyResetCount	Determines the number of non-encrypted bytes sent and received within an SSL conversation before the encryption key is renegotiated
StartStopEvent	Controls whether start and stop events are generated
SyncPoint	Syncpoint availability
TraceRouteRecording	Controls the recording of trace route information for messages
TreeLifeTime	The lifetime, in seconds, of non-administrative topics
TriggerInterval	Trigger-message interval
Notes: 1. This attribute cannot be inquired using the MQINQ call, and is not described in this section. For more information about this attribute, see Change Queue Manager .	

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

AuthorityEvent (10-digit signed integer)

Controls whether authorization (Not Authorized) events are generated.

The AuthorityEvent attribute must be set to one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAAUTE selector with the MQINQ call.

BridgeEvent (character string)

This attribute determines whether IMS bridge event messages are put onto the SYSTEM.ADMIN.CHANNEL.EVENT queue. It is only supported on z/OS.

ChannelAutoDef (10-digit signed integer)

Controls whether automatic channel definition is permitted.

This attribute controls the automatic definition of channels of type CTCRCVR and CTSVCN. Note that the automatic definition of CTCLSD channels is always enabled. This can have one of the following values:

CHADDI

Channel auto-definition disabled.

CHADEN

Channel auto-definition enabled.

To determine the value of this attribute, use the IACAD selector with the MQINQ call.

ChannelAutoDefEvent (10-digit signed integer)

Controls whether channel automatic-definition events are generated.

This applies to channels of type CTCVR, CTSVCN, and CTCLSD. This can have one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Monitoring and performance](#).

To determine the value of this attribute, use the IACADE selector with the MQINQ call.

ChannelAutoDefExit (20-byte character string)

Name of user exit for automatic channel definition.

If this name is nonblank, and *ChannelAutoDef* has the value CHADEN, the exit is called each time that the queue manager is about to create a channel definition. This applies to channels of type CTCVR, CTSVCN, and CTCLSD. The exit can then do one of the following:

- Allow the creation of the channel definition to proceed without change.
- Modify the attributes of the channel definition that is created.
- Suppress creation of the channel entirely.

To determine the value of this attribute, use the CACADX selector with the MQINQ call. The length of this attribute is given by LNEXTN.

ChannelEvent (character string)

Determines whether channel event messages are generated.

This attribute determines whether channel event messages are put onto the SYSTEM.ADMIN.CHANNEL.EVENT queue, and if so, what type of messages are queued (for example 'channel started', 'channel stopped', 'channel not activated'). Before the implementation of this attribute, the only way of preventing channel event messages from being queued was to delete the target queue.

This attribute also allows you to collect IMS bridge events only (because you can now switch off channel events, they do not get put onto the same queue). The same applies to SSL events which can also be collected without having to collect channel events as well.

This attribute also allows you to collect significant events only (for example when channels have errors, not when they start and stop normally).

The value for the ChannelEvent attribute can be one of the following:

- EVREXP (only the following channel events are generated: RC2279, RC2283, RC2284, RC2295, RC2296).
- EVRENA (all channel events are generated; that is, in addition to the events generated by EVREXP, the RC2282, and RC2283 events are also generated).
- EVRDIS (no channel events are generated; this is the queue manager initial default value).

To determine the value of this attribute, use the IACHNE selector with the MQINQ call.

ClusterCacheType (32-byte character string)

Controls whether cluster cache is fixed size, or is dynamically sized.

This is a user-defined 32-byte character string that is passed to the cluster workload exit when it is called. If there is no data to pass to the exit, the string is blank.

To determine the value of this attribute, use the CACLWD selector with the MQINQ call.

ClusterWorkloadData (32-byte character string)

User data for cluster workload exit.

This is a user-defined 32-byte character string that is passed to the cluster workload exit when it is called. If there is no data to pass to the exit, the string is blank.

To determine the value of this attribute, use the CACLWD selector with the MQINQ call.

ClusterWorkloadExit (20-byte character string)

Name of user exit for cluster workload management.

If this name is not blank, the exit is called each time that a message is put to a cluster queue or moved from one cluster-sender queue to another. The exit can then either accept the queue instance selected by the queue manager as the destination for the message, or select another queue instance.

To determine the value of this attribute, use the CACLWX selector with the MQINQ call. The length of this attribute is given by LNEXTN.

ClusterWorkloadLength (10-digit signed integer)

Maximum length of message data passed to cluster workload exit.

This is the maximum length of message data that is passed to the cluster workload exit. The actual length of data passed to the exit is the minimum of the following:

- The length of the message.
- The queue manager's *MaxMsgLength* attribute.
- The *ClusterWorkloadLength* attribute.

To determine the value of this attribute, use the IACLWL selector with the MQINQ call.

CodedCharSetId (10-digit signed integer)

Coded character set identifier.

This defines the character set used by the queue manager for all character string fields defined in the MQI such as the names of objects, and queue creation date and time. The character set must be one that has single-byte characters for the characters that are valid in object names. It does not apply to application data carried in the message. The value depends on the environment:

- On IBM i, the value is that which is set in the environment when the queue manager is first created.

To determine the value of this attribute, use the IACCSI selector with the MQINQ call.

CommandEvent (integer)

Controls whether messages are put onto a local queue when commands are issued.

This controls whether messages are written to a new event queue, SYSTEM.ADMIN.COMMAND.EVENT, whenever commands are issued. This feature is useful for command tracking notification, and for problem diagnosis. To inquire about the CommandEvent queue manager attribute, use the new attribute selector iacev with one of the following values:

- EVRENA - command event messages are generated and put onto the queue for all successful commands.
- EVND - command event messages are generated and put onto the queue for all successful commands other than the DISPLAY (MQSC) command, and the Inquire (PCF) command.
- EVRDIS - command event messages are not generated or put onto the queue (this is the queue manager's initial default value).

To determine the value of this attribute, use the CMDEV selector with the MQINQ call.

CommandInputQName (48-byte character string)

Command input queue name.

CommandInputQName is the name of the command input queue defined on the local queue manager. It is a queue to which users can send commands, if authorized to do so. The name of the queue depends on the environment:

- On IBM i, the name of the queue is SYSTEM.ADMIN.COMMAND.QUEUE, and only PCF commands can be sent to it. However, an MQSC command can be sent to this queue if the MQSC command is enclosed within a PCF command of type CMESC. For more information about the Escape command, see [Escape](#).

To determine the value of this attribute, use the CACMDQ selector with the MQINQ call. The length of this attribute is given by LNQN.

CommandLevel (10-digit signed integer)

Command Level. This indicates the level of system control commands supported by the queue manager.

The level is one of the following values:

CML700

Level 700 of system control commands.

This value is returned by the following applications:

- IBM MQ for IBM i Version 7 Release 0

CML701

Level 701 of system control commands.

This value is returned by the following applications:

- IBM MQ for IBM i Version 7 Release 0 Modification 1

CML71

Level 71 of system control commands.

This value is returned by the following applications:

- IBM MQ for IBM i Version 7 Release 1

CML800

Level 800 of system control commands.

This value is returned by the following applications:

- IBM MQ for IBM i
 - Version 8.0

The set of system control commands that corresponds to a particular value of the *CommandLevel* attribute varies according to the value of the *Platform* attribute; both must be used to decide which system control commands are supported.

To determine the value of this attribute, use the IACMDL selector with the MQINQ call.

ConfigurationEvent

Controls whether configuration events are generated and sent to the SYSTEM.ADMIN.CONFIG.EVENT queue default object.

The ConfigurationEvent attribute can be one of the following values:

- EVRENA
- EVRDIS

If the ConfigurationEvent attribute is set to EVRENA, and certain commands are successfully issued by runmqsc or PCF, configuration events are generated and sent to the SYSTEM.ADMIN.CONFIG.EVENT queue. Events for the following commands are issued, even if an alter command does not change the object involved. The commands for which configuration events are generated and sent are:

- DEFINE/ALTER AUTHINFO
- DEFINE/ALTER CHANNEL
- DEFINE/ALTER NAMELIST
- DEFINE/ALTER PROCESS
- DEFINE/ALTER QLOCAL (unless it is a temporary dynamic queue)
- DEFINE/ALTER QMODEL/QALIAS/QREMOTE
- DELETE AUTHINFO
- DELETE CHANNEL
- DELETE NAMELIST
- DELETE PROCESS
- DELETE QLOCAL (unless it is a temporary dynamic queue)
- DELETE QMODEL/QALIAS/QREMOTE
- ALTER QMGR (unless the CONFIGEV attribute is disabled and is not changed to enabled)
- REFRESH QMGR
- An MQSET call, other than for a temporary dynamic queue.

Events are not generated (if enabled) in the following circumstances:

- The command or MQSET call fails.
- The queue manager cannot put the event message on the event queue. The command should still complete successfully.
- Temporary dynamic queues.
- Internal attribute changes done directly or implicitly (not by MQSET or command); this affects TRIGGER, CURDEPTH, IPPROCS, OPPROCS, QDPHIEV, QDPLOEV, QDPMAXEV, QSVCI EV.
- When the configuration event queue is changed, although it an event message will be generated for that change when a Refresh is requested.
- Clustering changes by the commands REFRESH/RESET CLUSTER and RESUME/SUSPEND QMGR.
- Creating or deleting a queue manager.

DeadLetterQName (48-byte character string)

Name of dead-letter (undelivered-message) queue.

This is the name of a queue defined on the local queue manager. Messages are sent to this queue if they cannot be routed to their correct destination.

For example, messages are put on this queue when:

- A message arrives at a queue manager, destined for a queue that is not yet defined on that queue manager
- A message arrives at a queue manager, but the queue for which it is destined cannot receive it because, possibly:
 - The queue is full
 - Put requests are inhibited
 - The sending node does not have authority to put messages on the queue

Applications can also put messages on the dead-letter queue.

Report messages are treated in the same way as ordinary messages; if the report message cannot be delivered to its destination queue (typically the queue specified by the *MDRQ* field in the message descriptor of the original message), the report message is placed on the dead-letter (undelivered-message) queue.

Note: Messages that have passed their expiry time (see the *MDEXP* field described in “MQMD - Message descriptor” on page 1068) are **not** transferred to this queue when they are discarded. However, an expiration report message (ROEXP) is still generated and sent to the *MDRQ* queue, if requested by the sending application.

Messages are not put on the dead-letter (undelivered-message) queue when the application that issued the put request has been notified synchronously of the problem with the reason code returned by the MQPUT or MQPUT1 call (for example, a message put on a local queue for which put requests are inhibited).

Messages on the dead-letter (undelivered-message) queue sometimes have their application message data prefixed with an MQDLH structure. This structure contains extra information that indicates why the message was placed on the dead-letter (undelivered-message) queue. See “MQDLH - Dead-letter header” on page 1023 for more details of this structure.

This queue must be a local queue, with a *Usage* attribute of USNORM.

If a dead-letter (undelivered-message) queue is not supported by a queue manager, or one has not been defined, the name is all blanks. All IBM MQ queue managers support a dead-letter (undelivered-message) queue, but by default it is not defined.

If the dead-letter (undelivered-message) queue is not defined, or it is full, or unusable for some other reason, a message which would have been transferred to it by a message channel agent is retained instead on the transmission queue.

To determine the value of this attribute, use the CADLQ selector with the MQINQ call. The length of this attribute is given by LNQN.

DefClusterXmitQueueType (10-digit signed integer)

The DefClusterXmitQueueType attribute controls which transmission queue is selected by default by cluster-sender channels to get messages from, to send the messages to cluster-receiver channels.

The values of **DefClusterXmitQueueType** are MQCLXQ_SCTQ or MQCLXQ_CHANNEL.

MQCLXQ_SCTQ

All cluster-sender channels send messages from SYSTEM.CLUSTER.TRANSMIT.QUEUE. The correlID of messages placed on the transmission queue identifies which cluster-sender channel the message is destined for.

SCTQ is set when a queue manager is defined. This behavior is implicit in versions of IBM WebSphere MQ, earlier than Version 7.5. In earlier versions, the queue manager attribute DefClusterXmitQueueType was not present.

MQCLXQ_CHANNEL

Each cluster-sender channel sends messages from a different transmission queue. Each transmission queue is created as a permanent dynamic queue from the model queue SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE.

If the queue manager attribute, DefClusterXmitQueueType, is set to CHANNEL, the default configuration is changed to cluster-sender channels being associated with individual cluster transmission queues. The transmission queues are permanent-dynamic queues created from the model queue SYSTEM.CLUSTER.TRANSMIT.MODEL.QUEUE. Each transmission queue is associated with one cluster-sender channel. As one cluster-sender channel services a cluster transmission queue, the transmission queue contains messages for only one queue manager in one cluster. You can configure clusters so that each queue manager in a cluster contains only one cluster queue. In this case, the message traffic from a queue manager to each cluster queue is transferred separately from messages to other queues.

To query the value, call MQINQ, or send an Inquire Queue Manager (MQCMD_INQUIRE_Q_MGR) PCF command, setting the MQIA_DEF_CLUSTER_XMIT_Q_TYPE selector. To change the value, send a Change Queue Manager (MQCMD_CHANGE_Q_MGR) PCF command, setting the MQIA_DEF_CLUSTER_XMIT_Q_TYPE selector.

Related reference

[Change Queue Manager](#)

[Inquire Queue Manager](#)

[“MQINQ - Inquire about object attributes” on page 1261](#)

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object.

DefXmitQName (48-byte character string)

Default transmission queue name.

This is the name of the transmission queue that is used for the transmission of messages to remote queue managers, if there is no other indication of which transmission queue to use.

If there is no default transmission queue, the name is entirely blank. The initial value of this attribute is blank.

To determine the value of this attribute, use the CADXQN selector with the MQINQ call. The length of this attribute is given by LNQN.

DistLists (10-digit signed integer)

Distribution list support.

This indicates whether the local queue manager supports distribution lists on the MQPUT and MQPUT1 calls. This can have one of the following values:

DLSUPP

Distribution lists supported.

DLNSUP

Distribution lists not supported.

To determine the value of this attribute, use the IADIST selector with the MQINQ call.

InhibitEvent (10-digit signed integer)

Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated.

This can have one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Monitoring and performance](#).

To determine the value of this attribute, use the IAINHE selector with the MQINQ call.

LocalEvent (10-digit signed integer)

Controls whether local error events are generated.

The value is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#)

To determine the value of this attribute, use the IALCLE selector with the MQINQ call.

LoggerEvent (10-digit signed integer)

Controls whether recovery logger events are generated.

This can have one of the following values:

ENABLED

Logger events are generated.

DISABLED

Logger events are not generated. This is the queue managers initial default value.

For more information about events, see [Monitoring and performance](#).

MaxHandles (10-digit signed integer)

Maximum number of handles.

This is the maximum number of open handles that any one task can use concurrently. Each successful MQOPEN call for a single queue (or for an object that is not a queue) uses one handle. That handle becomes available for reuse when the object is closed. However, when a distribution list is opened, each queue in the distribution list is allocated a separate handle, and so that MQOPEN call uses as many handles as there are queues in the distribution list. This must be taken into account when deciding on a suitable value for *MaxHandles*.

The MQPUT1 call performs an MQOPEN call as part of its processing; as a result, MQPUT1 uses as many handles as MQOPEN would, but the handles are used only for the duration of the MQPUT1 call itself.

The value is in the range 1 through 999 999 999. On IBM i, the default value is 256.

To determine the value of this attribute, use the IAMHND selector with the MQINQ call.

MaxMsgLength (10-digit signed integer)

Maximum message length in bytes.

This is the length of the longest *physical* message that can be handled by the queue manager. However, because the *MaxMsgLength* queue manager attribute can be set independently of the *MaxMsgLength* queue attribute, the longest physical message that can be placed on a queue is the lesser of those two values.

If the queue manager supports segmentation, it is possible for an application to put a *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MFSEGA flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but typically, resource constraints imposed by the operating system or by the environment in which the application is running, will result in a lower limit.

The lower limit for the *MaxMsgLength* attribute is 32 KB (32 768 bytes). On IBM i, the maximum message length is 100 MB (104 857 600 bytes).

To determine the value of this attribute, use the IAMLEN selector with the MQINQ call.

MaxPriority (10-digit signed integer)

Maximum priority.

This is the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to *MaxPriority* (highest).

To determine the value of this attribute, use the IAMPRI selector with the MQINQ call.

MaxUncommittedMsgs (10-digit signed integer)

Maximum number of uncommitted messages within a unit of work.

This is the maximum number of uncommitted messages that can exist within a unit of work. The number of uncommitted messages is the sum of the following since the start of the current unit of work:

- Messages put by the application with the PMSYP option
- Messages retrieved by the application with the GMSYP option
- Trigger messages and COA report messages generated by the queue manager for messages put with the PMSYP option
- COD report messages generated by the queue manager for messages retrieved with the GMSYP option

The following are *not* counted as uncommitted messages:

- Messages put or retrieved by the application outside a unit of work
- Trigger messages or COA/COD report messages generated by the queue manager as a result of messages put or retrieved outside a unit of work
- Expiration report messages generated by the queue manager (even if the call causing the expiration report message specified GMSYP)
- Event messages generated by the queue manager (even if the call causing the event message specified PMSYP or GMSYP)

Note:

1. Exception report messages are generated by the Message Channel Agent (MCA), or by the application, and so are treated in the same way as ordinary messages put or retrieved by the application.
2. When a message or segment is put with the PMSYP option, the number of uncommitted messages is incremented by one regardless of how many physical messages actually result from the put. (More than one physical message might result if the queue manager needs to subdivide the message or segment.)
3. When a distribution list is put with the PMSYP option, the number of uncommitted messages is incremented by one *for each physical message that is generated*. This can be as small as one, or as great as the number of destinations in the distribution list.

The lower limit for this attribute is 1; the upper limit is 999 999 999.

To determine the value of this attribute, use the IAMUNC selector with the MQINQ call.

PerformanceEvent (10-digit signed integer)

Controls whether performance-related events are generated.

PerformanceEvent can have one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IAPFME selector with the MQINQ call.

Platform (10-digit signed integer)

Platform on which the queue manager is running.

This indicates the operating system on which the queue manager is running. The value is:

PL400

IBM i.

PubSubMode (10-digit signed integer)

Whether the publish/subscribe engine and the queued publish/subscribe interface are running, therefore allowing applications to publish/subscribe by using the application programming interface and the queues that are being monitored by the queued publish/subscribe interface.

This can have one of the following values:

PSMCP

The publish/subscribe engine is running. It is therefore possible to publish/subscribe by using the application programming interface. The queued publish/subscribe interface is not running, therefore any message that is put to the queues that are monitored by the queued publish/subscribe interface is not acted on. This setting is used for compatibility with WebSphere Message Broker V6 or earlier versions using this queue manager, because it must read the same queues from which the queued publish/subscribe interface normally reads.

PSMDS

The publish/subscribe engine and the queued publish/subscribe interface are not running. It is therefore not possible to publish/subscribe by using the application programming interface. Any publish/subscribe messages that are put to the queues that are monitored by the queued publish/subscribe interface are not acted on.

PSMEN

The publish/subscribe engine and the queued publish/subscribe interface are running. It is therefore possible to publish/subscribe by using the application programming interface and the queues that are being monitored by the queued publish/subscribe interface. This is the queue manager's initial default value.

To determine the value of this attribute, use the PSMODE selector with the MQINQ call.

QMGrDesc (64-byte character string)

Queue manager description.

This is a field that can be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager might require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

On IBM i, the default value is blanks.

To determine the value of this attribute, use the CAQMD selector with the MQINQ call. The length of this attribute is given by LNQMD.

QMGrIdentifier (48-byte character string)

Unique internally-generated identifier of queue manager.

This is an internally-generated unique name for the queue manager.

To determine the value of this attribute, use the CAQMID selector with the MQINQ call. The length of this attribute is given by LNQMID.

QMGrName (48-byte character string)

Queue manager name.

This is the name of the local queue manager, that is, the name of the queue manager to which the application is connected.

The first 12 characters of the name are used to construct a unique message identifier (see the *MDMID* field described in “MQMD - Message descriptor” on page 1068). Queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, in order for message identifiers to be unique in the queue manager network.

To determine the value of this attribute, use the CAQMN selector with the MQINQ call. The length of this attribute is given by LNQMN.

RemoteEvent (10-digit signed integer)

Controls whether remote error events are generated.

The value is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IARMTE selector with the MQINQ call.

RepositoryName (48-byte character string)

Name of cluster for which this queue manager provides repository services.

This is the name of a cluster for which this queue manager provides a repository-manager service. If the queue manager provides this service for more than one cluster, *RepositoryNameList* specifies the name of a namelist object that identifies the clusters, and *RepositoryName* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

To determine the value of this attribute, use the CARPN selector with the MQINQ call. The length of this attribute is given by LNQMNM.

RepositoryNameList (48-byte character string)

Name of namelist object containing names of clusters for which this queue manager provides repository services.

This is the name of a namelist object that contains the names of clusters for which this queue manager provides a repository-manager service. If the queue manager provides this service for only one cluster, the namelist object contains only one name. Alternatively, *RepositoryName* can be used to specify the name of the cluster, in which case *RepositoryNameList* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

To determine the value of this attribute, use the CARPNL selector with the MQINQ call. The length of this attribute is given by LNNLN.

SSLEvent (character string)

Determines whether SSL events are generated.

The value is one of the following:

- EVRENA (MQINQ/PCF/config event) ENABLED (MQSC): SSL events are generated (that is, the RC2371 event is generated).
- EVRDIS (MQINQ/PCF/config event) DISABLED (MQSC): SSL events are not generated. This is the queue manager's initial default value.

To determine the value of this attribute, use the IASSLE selector with the MQINQ call.

SSLKeyResetCount (integer)

Determines the total number of non-encrypted bytes that are sent and received within an SSL conversation, before the secret key is renegotiated. The number of bytes includes control information sent by the message channel agent (MCA).

This value is only used by SSL channel MCAs which initiate communication from this queue manager (that is, the sender channel MCA in a sender and receiver channel pairing).

If the value of this attribute is greater than 0, and channel heartbeats are enabled for a channel, the secret key is also renegotiated before data is sent or received following a channel heartbeat. The count of bytes until the next secret key renegotiation is reset after each successful renegotiation occurs.

The value can be in the range 0 through 999 999 999. A value of 0 for this attribute indicates that the secret key is never renegotiated. If you specify an SSL/TLS secret key reset count in the range 1 byte through 32 KB, SSL/TLS channels will use a secret key reset count of 32 KB. This is to avoid the processing cost of excessive key resets which would occur for small SSL/TLS secret key reset values.

When the SSL server is an IBM MQ queue manager, and both secret key reset and channel heartbeats are enabled, renegotiation occurs immediately after each channel heartbeat.

To determine the value of this attribute, use the IASSRC selector with the MQINQ call.

StartStopEvent (10-digit signed integer)

Controls whether start and stop events are generated.

This attribute can have one of the following values:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

For more information about events, see [Event monitoring](#).

To determine the value of this attribute, use the IASSE selector with the MQINQ call.

SyncPoint (10-digit signed integer)

Syncpoint availability.

This indicates whether the local queue manager supports units of work and syncpointing with the MQGET, MQPUT, and MQPUT1 calls.

SPAVL

Units of work and syncpointing available.

SPNAVL

Units of work and syncpointing not available.

To determine the value of this attribute, use the IASYNC selector with the MQINQ call.

TraceRouteRecording (10-digit signed integer)

This controls whether information about messages is recorded as they flow through a queue manager.

The value is one of the following:

- RECDD: no appending to trace route messages is allowed
- RECDQ: messages are put onto a fixed named queue
- RECDM: determine using message (this is the initial default setting)

To prevent the trace route message from remaining in the system, set an expiry value on it that is greater than zero, and specify the RODISC report option. To prevent report or reply messages remaining in the system, set the report option ROPDAE. For more information, see [“Report options and message flags” on page 1388](#).

To determine the value of this attribute, use the IATRGI selector with the MQINQ call.

TreeLifeTime (10-digit signed integer)

The lifetime, in seconds, of non-administrative topics.

Non-administrative topics are those created when an application publishes to, or subscribes as, a topic string that does not exist as an administrative node. When this non-administrative node no longer has any active subscriptions, this parameter determines how long the queue manager will wait before removing that node. Only non-administrative topics that are in use by a durable subscription remain after the queue manager is recycled.

Specify a value in the range 0 through 604 000. A value of 0 means that non-administrative topics are not removed by the queue manager. The queue manager's initial default value is 1800.

To determine the value of this attribute, use the IATRLFT selector with the MQINQ call.

TriggerInterval (10-digit signed integer)

Trigger-message interval.

This is a time interval (in milliseconds) used to restrict the number of trigger messages. This is relevant only when the *TriggerType* is TTFRST. In this case, trigger messages are normally generated only when a suitable message arrives on the queue, and the queue was previously empty. Under certain circumstances, however, an additional trigger message can be generated with TTFRST triggering even if the queue was not empty. These additional trigger messages are not generated more often than every *TriggerInterval* milliseconds.

For more information about triggering, see [Triggering channels](#).

The value is in the range zero through 999 999 999. The default value is 999 999 999.

To determine the value of this attribute, use the IATRGI selector with the MQINQ call.

Applications

This information describes the sample programs delivered with IBM MQ for IBM i for RPG. Also, learn how to build executable applications from the programs you write.

Building your application

The IBM i publications describe how to build executable applications from the programs you write. This topic describes the additional tasks, and the changes to the standard tasks, you must perform when building IBM MQ for IBM i applications to run under IBM i.

In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the IBM MQ for IBM i copy files for the RPG language. You should make yourself familiar with the contents of these files; their names, and a brief description of their contents are given in the following text.

IBM MQ copy files

IBM MQ for IBM i provides copy files to assist you with writing your applications in the RPG programming language. They are suitable for use with the WebSphere Development toolset (5722 WDS) ILE RPG 4 Compiler.

The copy files that IBM MQ for IBM i provides to assist with the writing of channel exits are described in [Channel-exit programs for messaging channels](#).

The names of the IBM MQ for IBM i copy files for RPG have the prefix CMQ. They have a suffix of G or H. There are separate copy files containing the named constants, and one file for each of the structures. The copy files are listed in [“Language considerations” on page 971](#).

Note: For ILE RPG/400, they are supplied as members of file QRPGLSRC in library QMQM.

The structure declarations do not contain DS statements. This allows the application to declare a data structure (or a multiple-occurrence data structure) by coding the DS statement and using the /COPY statement to copy in the remainder of the declaration:

For ILE RPG/400 the statement is:

```
D*..1.....2.....3.....4.....5.....6.....7
D* Declare an MQMD data structure
D MQMD          DS
D/COPY CMQMDG
```

Preparing your programs to run

To create an executable IBM MQ for IBM i application, you have to compile the source code you have written.

To do this for ILE RPG/400, you can use the typical IBM i commands, CRTRPGMOD and CRTPGM.

After creating your *MODULE, you need to specify BNDSRVPGM (QMQM/LIBMQM) in the CRTPGM command. This includes the various IBM MQ procedures in your program.

Make sure that the library containing the copy files (QMQM) is in the library list when you perform the compilation.

For further information concerning programming considerations, including client modes, see [“Language considerations” on page 971](#).

Interfaces to the IBM i external syncpoint manager

IBM MQ for IBM i uses native IBM i commitment control as an external syncpoint coordinator.

See the *IBM i Programming: Backup and Recovery Guide* for more information about the commitment control capabilities of IBM i.

To start the IBM i commitment control facilities, use the STRCMTCTL system command. To end commitment control, use the ENDCMTCTL system command.

Note: The default value of *Commitment definition scope* is *ACTGRP. This must be defined as *JOB for IBM MQ for IBM i. For example:

```
STRCMTCTL LCKLVL(*ALL) CMTSCOPE(*JOB)
```

If you call MQPUT, MQPUT1, or MQGET, specifying PMSYP or GMSYP, after starting commitment control, IBM MQ for IBM i adds itself as an API commitment resource to the commitment definition. This is typically the first such call in a job. While there are any API commitment resources registered under a particular commitment definition, you cannot end commitment control for that definition.

IBM MQ for IBM i removes its registration as an API commitment resource when you disconnect from the queue manager, provided there are no pending MQI operations in the current unit of work.

If you disconnect from the queue manager while there are pending MQPUT, MQPUT1, or MQGET operations in the current unit of work, IBM MQ for IBM i remains registered as an API commitment resource so that it is notified of the next commit or rollback. When the next syncpoint is reached, IBM MQ commits or rolls back the changes as required. It is possible for an application to disconnect and reconnect to a queue manager during an active unit of work and perform further MQGET and MQPUT operations inside the same unit of work (this is a pending disconnect).

If you attempt to issue an ENDCMTCTL system command for that commitment definition, message CPF8355 is issued, indicating that pending changes were active. This message also appears in the job log when the job ends. To avoid this, ensure that you commit or roll back all pending IBM MQ operations, and that you disconnect from the queue manager. Thus, using COMMIT or ROLLBACK commands before ENDCMTCTL should enable end-commitment control to complete successfully.

When IBM i commitment control is used as an external syncpoint coordinator, MQCMIT, MQBACK, and MQBEGIN calls might not be issued. Calls to these functions fail with the reason code RC2012.

To commit or roll back (that is, to back out) your unit of work, use one of the programming languages that supports the commitment control. For example:

- CL commands: COMMIT and ROLLBACK
- ILE C Programming Functions: _Rcommit and _Rrollback
- RPG/400: COMMIT and ROLBK
- COBOL/400: COMMIT and ROLLBACK

Syncpoints in CICS for IBM i applications

IBM MQ for IBM i participates in units of work with CICS. You can use the MQI within a CICS application to put and get messages inside the current unit of work.

You can use the EXEC CICS SYNCPOINT command to establish a syncpoint that includes the IBM MQ for IBM i operations. To back out all changes up to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command.

If you use MQPUT, MQPUT1, or MQGET with the PMSYP, or GMSYP, option set in a CICS application, you cannot log off CICS until IBM MQ for IBM i has removed its registration as an API commitment resource. Therefore, you should commit or back out any pending put or get operations before you disconnect from the queue manager. This will allow you to log off CICS.

Sample programs on IBM i

This topic describes the sample programs delivered with IBM MQ for IBM i for RPG. The samples demonstrate typical uses of the Message Queue Interface (MQI).

The samples are not intended to demonstrate general programming techniques, so some error checking that you may want to include in a production program has been omitted. However, these samples are suitable for use as a base for your own message queuing programs.

The source code for all the samples is provided with the product; this source includes comments that explain the message queuing techniques demonstrated in the programs.

There is one set of ILE sample programs:

1. Programs using prototyped calls to the MQI (static bound calls)

The source exists in QMQMSAMP/QRPGLESRC. The members are named AMQ3xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name has a suffix of G or H.

Table 628 on page 1371 gives a complete list of the sample programs delivered with IBM MQ for IBM i, and shows the names of the programs in each of the supported programming languages. Notice that their names all start with the prefix AMQ, the fourth character in the name indicates the programming language.

Table 628. Names of the sample programs	
	RPG (ILE)
Put samples	AMQ3PUT4
Browse samples	AMQ3GBR4
Get samples	AMQ3GET4
Request samples	AMQ3REQ4
Echo samples	AMQ3ECH4
Inquire samples	AMQ3INQ4
Set samples	AMQ3SET4
Trigger Monitor sample	AMQ3TRG4
Trigger Server sample	AMQ3SRV4

In addition to these, the IBM MQ for IBM i sample option includes a sample data file, AMQSDATA, which can be used as input to certain sample programs and sample CL programs that demonstrate administration tasks. The CL samples are described in [Administering IBM i](#). You could use the sample CL program to create queues to use with the sample programs described in this topic.

For information about how to run the sample programs, see [“Preparing and running the sample programs on IBM i”](#) on page 1372.

Features demonstrated in the sample programs on IBM i

A table that shows the techniques demonstrated by the IBM MQ for IBM i sample programs.

Some techniques occur in more than one sample program, but only one program is listed in the table. All the samples open and close queues using the MQOPEN and MQCLOSE calls, so these techniques are not listed separately in the table.

<i>Table 629. Sample programs demonstrating use of the MQI</i>	
Technique	RPG (ILE)
Using the MQCONN and MQDISC calls	AMQ3ECH4 or AMQ3INQ4
Implicitly connecting and disconnecting	AMQ3PUT4
Putting messages using the MQPUT call	AMQ3PUT4
Putting a single message using the MQPUT1 call	AMQ3ECH4 or AMQ3INQ4
Replying to a request message	AMQ3INQ4
Getting messages (no wait)	AMQ3GBR4
Getting messages (wait with a time limit)	AMQ3GET4
Getting messages (with data conversion)	AMQ3ECH4
Browsing a queue	AMQ3GBR4
Using a shared input queue	AMQ3INQ4
Using an exclusive input queue	AMQ3REQ4
Using the MQINQ call	AMQ3INQ4
Using the MQSET call	AMQ3SET4
Using a reply-to queue	AMQ3REQ4
Requesting exception messages	AMQ3REQ4
Accepting a truncated message	AMQ3GBR4
Using a resolved queue name	AMQ3GBR4
Trigger processing	AMQ3SRV4 or AMQ3TRG4

Note: All the sample programs produce a spool file that contains the results of the processing.

Preparing and running the sample programs on IBM i

Before you can run the IBM MQ for IBM i sample programs, you must compile them as you would any other IBM MQ for IBM i applications. To do so, you can use the IBM i commands CRTRPGMOD and CRTPGM.

When you create the AMQ3xxx4 programs, you must specify BNDSRVPGM(QMQM/LIBMQM) in the CRTPGM command. Doing so includes the various IBM MQ procedures in your program.

The sample programs are provided in library QMQMSAMP as members of QRPGLSRC. They use the copy files provided in library QMQM, so make sure that this library is in the library list when you compile them. The RPG compiler gives information messages because the samples do not use many of the variables that are declared in the copy files.

Running the sample programs

You can use your own queues when you run the samples, or you can compile and run AMQSAMP4 to create some sample queues. The source for this program is shipped in file QCLSRC in library QMQMSAMP. It can be compiled using the CRTCLPGM command.

To call one of the sample programs, use a command like:

```
CALL PGM(QMQMSAMP/AMQ3PUT4) PARM('Queue_Name','Queue_Manager_Name')
```

Where `Queue_Name` and `Queue_Manager_Name` must be 48 characters in length, which you achieve by padding the `Queue_Name` and `Queue_Manager_Name` with the required number of blanks.

For the Inquire and Set sample programs, the sample definitions created by AMQSAMP4 cause the C versions of these samples to be triggered. If you want to trigger the RPG versions, you must change the process definitions `SYSTEM.SAMPLE.ECHOPROCESS` and `SYSTEM.SAMPLE.INQPROCESS` and `SYSTEM.SAMPLE.SETPROCESS`. You can use the `CHGMQMPRC` command (described in [Change MQ Process \(CHGMQMPRC\)](#)) to do so, or edit and run AMQSAMP4 with the alternative definition.

The Put sample program on IBM i

The Put sample program, AMQ3PUT4, puts messages on a queue using the MQPUT call.

To start the program, call the program and give the name of your target queue as a program parameter. The program puts a set of fixed messages on the queue; these messages are taken from the data block at the end of the program source code. A sample put program is AMQ3PUT4 in library QMQMSAMP.

Using this example program, the command is:

```
CALL PGM(QMQMSAMP/AMQ3PUT4) PARM('Queue_Name','Queue_Manager_Name')
```

Where `Queue_Name` and `Queue_Manager_Name` *must* be 48 characters in length, which you achieve by padding the `Queue_Name` and `Queue_Manager_Name` with the required number of blanks.

Design of the Put sample program

The program uses the MQOPEN call with the OOOUT option to open the target queue for putting messages. The results are output to a spool file. If it cannot open the queue, the program writes an error message containing the reason code returned by the MQOPEN call. To keep the program simple, on this and on subsequent MQI calls, the program uses default values for many of the options.

For each line of data contained in the source code, the program reads the text into a buffer and uses the MQPUT call to create a datagram message containing the text of that line. The program continues until either it reaches the end of the input or the MQPUT call fails. If the program reaches the end of the input, it closes the queue using the MQCLOSE call.

The Browse sample program on IBM i

The Browse sample program, AMQ3GBR4, browses messages on a queue using the MQGET call.

The program retrieves copies of all the messages on the queue you specify when you call the program; the messages remain on the queue. You could use the supplied queue `SYSTEM.SAMPLE.LOCAL`; run the Put sample program first to put some messages on the queue. You could use the queue `SYSTEM.SAMPLE.ALIAS`, which is an alias name for the same local queue. The program continues until it reaches the end of the queue or an MQI call fails.

An example of a command to call the RPG program is:

```
CALL PGM(QMQMSAMP/AMQ3GBR4) PARM('Queue_Name','Queue_Manager_Name')
```

Where `Queue_Name` and `Queue_Manager_Name` *must* be 48 characters in length, which you achieve by padding the `Queue_Name` and `Queue_Manager_Name` with the required number of blanks. Therefore, if you are using `SYSTEM.SAMPLE.LOCAL` as your target queue, you will need 29 blank characters.

Design of the Browse sample program

The program opens the target queue using the MQOPEN call with the OOBROW option. If it cannot open the queue, the program writes an error message to its spool file, containing the reason code returned by the MQOPEN call.

For each message on the queue, the program uses the MQGET call to copy the message from the queue, then displays the data contained in the message. The MQGET call uses these options:

GMBRWN

After the MQOPEN call, the browse cursor is positioned logically before the first message in the queue, so this option causes the *first* message to be returned when the call is first made.

GMNWT

The program does not wait if there are no messages on the queue.

GMATM

The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the program displays the truncated message, together with a warning that the message has been truncated.

The program demonstrates how you must clear the *MDMID* and *MDCID* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The program continues to the end of the queue; here, the MQGET call returns the RC2033 (no message available) reason code and the program displays a warning message. If the MQGET call fails, the program writes an error message that contains the reason code in its spool file.

The program then closes the queue using the MQCLOSE call.

The Get sample program on IBM i

The Get sample program, AMQ3GET4, gets messages from a queue using the MQGET call.

When the program is called, it removes messages from the specified queue. You could use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You could use the SYSTEM.SAMPLE.ALIAS queue, which is an alias name for the same local queue. The program continues until the queue is empty or an MQI call fails.

An example of a command to call the RPG program is:

```
CALL PGM(QMQMSAMP/AMQ3GET4) PARM('Queue_Name','Queue_Manager_Name')
```

where Queue_Name and Queue_Manager_Name *must* be 48 characters in length, which you achieve by padding the Queue_Name and Queue_Manager_Name with the required number of blanks. Therefore, if you are using SYSTEM.SAMPLE.LOCAL as your target queue, you will need 29 blank characters.

Design of the Get sample program

The program opens the target queue for getting messages; it uses the MQOPEN call with the OOINPQ option. If it cannot open the queue, the program writes an error message containing the reason code returned by the MQOPEN call in its spool file.

For each message on the queue, the program uses the MQGET call to remove the message from the queue; it then displays the data contained in the message. The MQGET call uses the GMWT option, specifying a wait interval (*GMWT*) of 15 seconds, so that the program waits for this period if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the RC2033 (no message available) reason code.

The program demonstrates how you must clear the *MDMID* and *MDCID* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the call fails and the program stops.

The program continues until either the MQGET call returns the RC2033 (no message available) reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

The program then closes the queue using the MQCLOSE call.

The Request sample program on IBM i

The Request sample program, AMQ3REQ4, demonstrates client/server processing. The sample is the client that puts request messages on a queue that is processed by a server program. It waits for the server program to put a reply message on a reply-to queue.

The Request sample puts a series of request messages on a queue using the MQPUT call. These messages specify SYSTEM.SAMPLE.REPLY as the reply-to queue. The program waits for reply messages, then displays them. Replies are sent only if the target queue (which we will call the *server queue*) is being processed by a server application, or if an application is triggered for that purpose (the Inquire and Set sample programs are designed to be triggered). The sample waits 5 minutes for the first reply to arrive (to allow time for a server application to be triggered) and 15 seconds for subsequent replies, but it can end without getting any replies.

To start the program, call the program and give the name of your target queue as a program parameter. The program puts a set of fixed messages on the queue; these messages are taken from the data block at the end of the program source code.

Design of the Request sample program

The program opens the server queue so that it can put messages. It uses the MQOPEN call with the OOOOUT option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

The program then opens the reply-to queue called SYSTEM.SAMPLE.REPLY so that it can get reply messages. For this, the program uses the MQOPEN call with the OOINPX option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

For each line of input, the program then reads the text into a buffer and uses the MQPUT call to create a request message containing the text of that line. On this call the program uses the ROEXCD report option to request that any report messages sent about the request message will include the first 100 bytes of the message data. The program continues until either it reaches the end of the input or the MQPUT call fails.

The program then uses the MQGET call to remove reply messages from the queue, and displays the data contained in the replies. The MQGET call uses the GMWT option, specifying a wait interval (*GMWT*) of 5 minutes for the first reply (to allow time for a server application to be triggered) and 15 seconds for subsequent replies. The program waits for these periods if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the RC2033 (no message available) reason code. The call also uses the GMATM option, so messages longer than the declared buffer size are truncated.

The program demonstrates how you must clear the *MDMID* and *MDCOD* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The program continues until either the MQGET call returns the RC2033 (no message available) reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

The program then closes both the server queue and the reply-to queue using the MQCLOSE call. Table 630 on page 1375 shows the changes to the Echo sample program that are necessary to run the Inquire and Set sample programs.

Note: The details for the Echo sample program are included as a reference.

<i>Table 630. Client/Server sample program details</i>		
Program name	SYSTEM/SAMPLE queue	Program started
Echo	ECHO	AMQ3ECH4
Inquire	INQ	AMQ3INQ4

Table 630. Client/Server sample program details (continued)		
Program name	SYSTEM/SAMPLE queue	Program started
Set	SET	AMQ3SET4

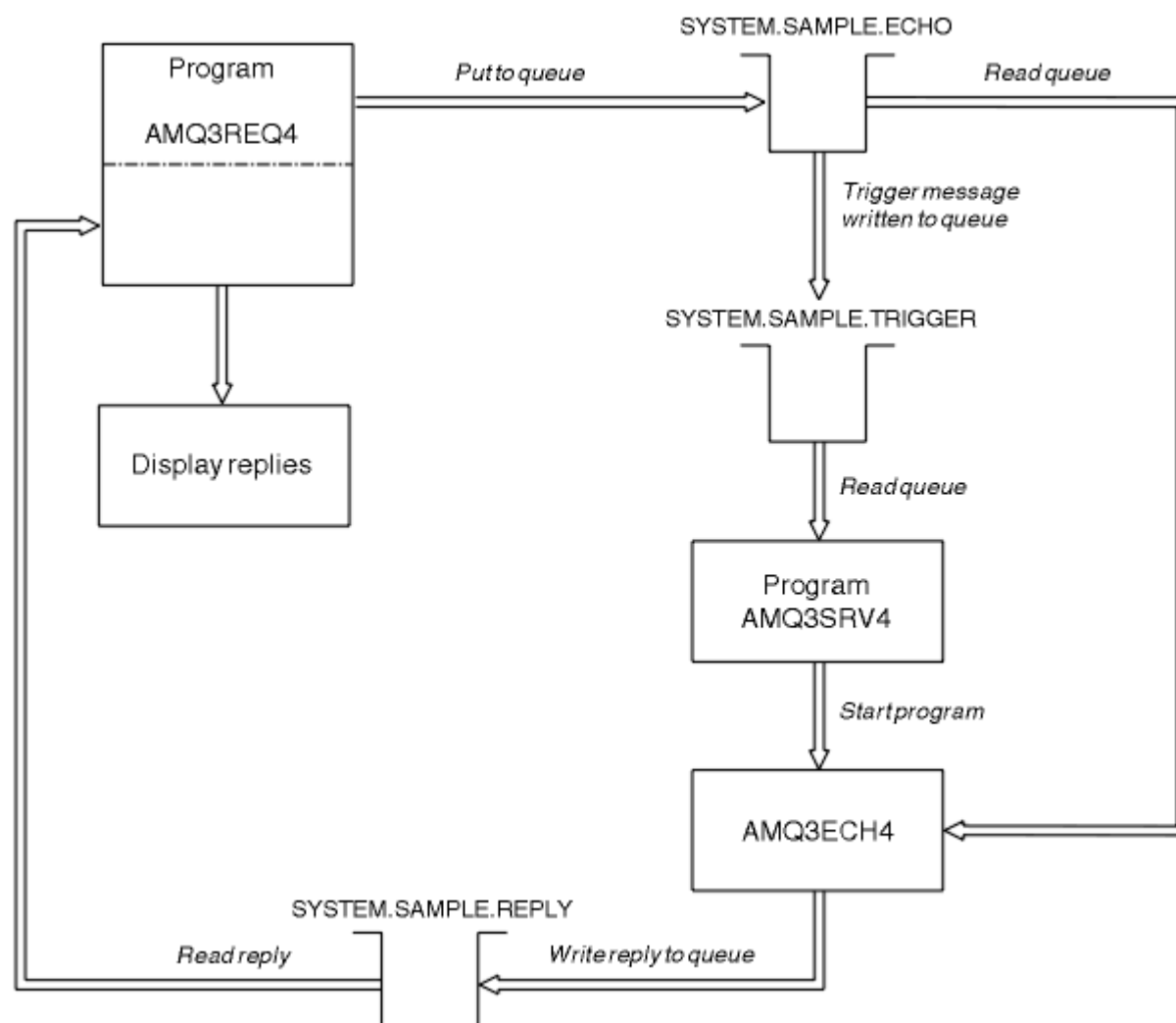


Figure 9. Sample Client/Server (Echo) program flowchart

Using triggering with the Request sample

To run the sample using triggering, start the trigger server program, AMQ3SRV4, against the required initiation queue in one job, then start AMQ3REQ4 in another job.

This means that the trigger server is ready when the Request sample program sends a message.

Note:

1. The samples use the SYSTEM SAMPLE TRIGGER queue as the initiation queue for SYSTEM.SAMPLE.ECHO, SYSTEM.SAMPLE.INQ, or SYSTEM.SAMPLE.SET local queues. Alternatively, you can define your own initiation queue.
2. The sample definitions created by AMQSAMP4 cause the C version of the sample to be triggered. If you want to trigger the RPG version, you must change the process definitions SYSTEM.SAMPLE.ECHOPROCESS and SYSTEM.SAMPLE.INQPROCESS and SYSTEM.SAMPLE.SETPROCESS. You can use the CHGMQMPCRC command (see [Change MQ Process \(CHGMQMPCRC\)](#) for more details) to do this, or edit and run your own version of AMQSAMP4.
3. You must compile the trigger server program from the source provided in QMQMSAMP/QRPGLESRC.

Depending on the trigger process you want to run, AMQ3REQ4 should be called with the parameter specifying request messages to be placed on one of these sample server queues:

- SYSTEM.SAMPLE.ECHO (for the Echo sample programs)
- SYSTEM.SAMPLE.INQ (for the Inquire sample programs)
- SYSTEM.SAMPLE.SET (for the Set sample programs)

A flow chart for the SYSTEM.SAMPLE.ECHO program is shown in [Figure 9 on page 1376](#). Using the example the command to issue the RPG program request to this server is:

```
CALL PGM(QMQMSAMP/AMQ3REQ4) PARM('SYSTEM.SAMPLE.ECHO  
+ 30 blank characters','Queue_Manager_Name')
```

because the queue name and queue manager name *must* be 48 characters in length.

Note: This sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, server applications are not triggered by the messages you send.

If you want to attempt further examples, you can try the following variations:

- Use AMQ3TRG4 instead of AMQ3SRV4 to submit the job instead, but potential job submission delays could make it less easy to follow what is happening.
- Use the SYSTEM.SAMPLE.INQ and SYSTEM.SAMPLE.SET sample queues. Using the example data file, the commands to issue the RPG program requests to these servers are:

```
CALL PGM(QMQMSAMP/AMQ3INQ4) PARM('SYSTEM.SAMPLE.INQ  
+ 31 blank characters')  
CALL PGM(QMQMSAMP/AMQ3SET4) PARM('SYSTEM.SAMPLE.SET  
+ 31 blank characters')
```

because the queue name *must* be 48 characters in length.

These sample queues also have a trigger type of FIRST.

The Echo sample program on IBM i

The Echo sample programs return the message send to a reply queue. The program is named AMQ3ECH4

For the triggering process to work, you must ensure that the Echo sample program you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.ECHO. To do this, specify the name of the Echo sample program you want to use in the *AppLId* field of the process definition SYSTEM.SAMPLE.ECHOPROCESS. (For this, you can use the CHGMQMPCRC command, described in [Administering IBM i](#).) The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Echo sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQ3REQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send messages to queue SYSTEM.SAMPLE.ECHO. The Echo sample programs send a reply message containing the data in the request message to the reply-to queue specified in the request message.

Design of the Echo sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although this is not necessary for IBM MQ for IBM i, this means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the contents of the request message.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

This program can also respond to messages sent to the queue from platforms other than IBM MQ for IBM i, although no sample is supplied for this situation. To make the ECHO program work, you:

- Write a program, correctly specifying the *Format*, *Encoding*, and *CCSID* fields, to send text request messages.

The ECHO program requests the queue manager to perform message data conversion, if this is needed.

- Specify CONVERT(*YES) on the IBM MQ for IBM i sending channel, if the program you have written does not provide similar conversion for the reply.

The Inquire sample program on IBM i

The Inquire sample program, AMQ3INQ4, inquires about some of the attributes of a queue using the MQINQ call.

The program is intended to run as a triggered program, so its only input is an MQTMC (trigger message) structure. This structure contains the name of a target queue with attributes that are to be inquired upon.

For the triggering process to work, you must ensure that the Inquire sample program is triggered by messages arriving on queue SYSTEM.SAMPLE.INQ. To do so, specify the name of the Inquire sample program in the *AppId* field of the SYSTEM.SAMPLE.INQPROCESS process definition. (For this, you can use the CHGMQMPCRC command, described in [Change MQ Process \(CHGMQMPCRC\)](#)). The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Inquire sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQ3REQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays might make it less easy to follow what is happening.

Use the Request sample program to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.INQ. For each request message, the Inquire sample program sends a reply message containing information about the queue specified in the request message. The replies are sent to the reply-to queue specified in the request message.

Design of the Inquire sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although not necessary for IBM MQ for IBM i, this design feature means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message, and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data and opens that queue using the MQOPEN call with the OOINQ option. The program then uses the MQINQ call to inquire about the values of the *InhibitGet*, *CurrentQDepth*, and *OpenInputCount* attributes of the target queue.

If the MQINQ call is successful, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the values of the three attributes.

If the MQOPEN or MQINQ call is unsuccessful, the program uses the MQPUT call to put a *report* message on the reply-to queue. In the *MDFB* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQINQ call, depending on which one failed.

After the MQINQ call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Set sample program on IBM i

The Set sample program, AMQ3SET4, inhibits put operations on a queue by using the MQSET call to change the queue's *InhibitPut* attribute.

The program is intended to run as a triggered program, so its only input is an MQTMC (trigger message) structure that contains the name of a target queue with attributes that are to be inquired upon.

For the triggering process to work, you must ensure that the Set sample program is triggered by messages arriving on queue SYSTEM.SAMPLE.SET. To do this, specify the name of the Set sample program in the *AppLId* field of the process definition SYSTEM.SAMPLE.SETPROCESS. (For this, you can use the CHGMQMPPRC command, described in the *Administering IBM i*.) The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Set sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQ3REQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample program to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.SET. For each request message, the Set sample program sends a reply message containing a confirmation that put operations have been inhibited on the specified queue. The replies are sent to the reply-to queue specified in the request message.

Design of the Set sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although this is not necessary for IBM MQ for IBM i, this means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data and opens that queue using the MQOPEN call with the OOSSET option. The program then uses the MQSET call to set the value of the *InhibitPut* attribute of the target queue to QAPUTI.

If the MQSET call is successful, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the string PUT inhibited.

If the MQOPEN or MQSET call is unsuccessful, the program uses the MQPUT call to put a *report* message on the reply-to queue. In the *MDFB* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQSET call, depending on which one failed.

After the MQSET call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Triggering sample programs on IBM i

IBM MQ for IBM i supplies two Triggering sample programs that are written in ILE/RPG.

The programs are:

AMQ3TRG4

This is a trigger monitor for the IBM i environment. It submits an IBM i job for the application to be started, but this means that there is additional processing cost associated with each trigger message.

AMQ3SRV4

This is a trigger server for the IBM i environment. For each trigger message, this server runs the start command in its own job to start the specified application. The trigger server can call CICS transactions.

C language versions of these samples are also available as executable programs in library QMQM, called AMQSTRG4 and AMQSERV4.

The AMQ3TRG4 sample trigger monitor on IBM i

AMQ3TRG4 is a trigger monitor. It takes one parameter: the name of the initiation queue it is to serve. AMQSAMP4 defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

AMQ3TRG4 submits an IBM i job for each valid trigger message it gets from the initiation queue.

Design of the trigger monitor

The trigger monitor opens the initiation queue and gets messages from the queue, specifying an unlimited wait interval.

The trigger monitor submits an IBM i job to start the application specified in the trigger message, and passes an MQTMC (a character version of the trigger message) structure. The environment data in the trigger message is used as job submission parameters.

Finally, the program closes the initiation queue.

The AMQ3SRV4 sample trigger server

AMQ3SRV4 is a trigger server. It takes one parameter: the name of the initiation queue it is to serve. AMQSAMP4 defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

For each trigger message, AMQ3SRV4 runs a start command in its own job to start the specified application.

Using the example trigger queue the command to issue is:

```
CALL PGM(QMQM/AMQ3SRV4) PARM('Queue Name')
```

Where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks. Therefore, if you are using SYSTEM.SAMPLE.TRIGGER as your target queue, you will need 28 blank characters.

Design of the trigger server

The design of the trigger server is like that of the trigger monitor, except the trigger server:

- Allows CICS as well as IBM i applications
- Does not use the environment data from the trigger message
- Calls IBM i applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an IBM i job
- Opens the initiation queue for shared input, so many trigger servers can run at the same time

Note: Programs started by AMQ3SRV4 must not use the MQDISC call because this will stop the trigger server. If programs started by AMQ3SRV4 use the MQCONN call, they will get the RC2002 reason code.

Ending the Triggering sample programs on IBM i

A trigger monitor program can be ended by the sysrequest option 2 (ENDRQS) or by inhibiting gets from the trigger queue.

If the sample trigger queue is used the command is:

```
CHGMQM QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*NO)
```

Note: To start triggering again on this queue, you *must* enter the command:

```
CHGMQM QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*YES)
```

Running the samples using remote queues on IBM i

You can demonstrate remote queuing by running the samples on connected message queue managers.

Program AMQSAMP4 provides a local definition of a remote queue (SYSTEM.SAMPLE.REMOTE) that uses a remote queue manager named OTHER. To use this sample definition, change OTHER to the name of the second message queue manager you want to use. You must also set up a message channel between your two message queue managers; for information about how to do so, see [Channel-exit programs for messaging channels](#).

The Request sample program puts its own local queue manager name in the *MDRM* field of messages it sends. The Inquire and Set samples send reply messages to the queue and message queue manager named in the *MDRQ* and *MDRM* fields of the request messages they process.

Return codes for IBM i (ILE RPG)

This information describes the return codes associated with the MQI and MQAI.

The return codes associated with:

- Programmable Command Format (PCF) commands are listed in [Programmable command formats reference](#).
- C++ calls are listed in [Using C++](#).

For each call, a completion code and a reason code are returned by the queue manager or by an exit routine, to indicate the success or failure of the call.

Applications must not depend upon errors being checked for in a specific order, except where specifically noted. If more than one completion code or reason code could arise from a call, the particular error reported depends on the implementation.

Completion codes for IBM i (ILE RPG)

The completion code parameter (*CMPCOD*) allows the caller to see quickly whether the call completed successfully, completed partially, or failed.

CCOK

(MQCC_OK on other platforms)

Successful completion.

The call completed fully; all output parameters have been set. The *REASON* parameter always has the value RCNONE in this case.

CCWARN

(MQCC_WARN on other platforms)

Warning (partial completion).

The call completed partially. Some output parameters might have been set in addition to the *CMPCOD* and *REASON* output parameters. The *REASON* parameter gives additional information about the partial completion.

CCFAIL

(MQCC_FAIL on other platforms)

Call failed.

The processing of the call did not complete, and the state of the queue manager is normally unchanged; exceptions are specifically noted. The *CMPCOD* and *REASON* output parameters have been set; other parameters are unchanged, except where noted.

The reason might be a fault in the application program, or it might be a result of some situation external to the program, for example the user's authority might have been revoked. The *REASON* parameter gives additional information about the error.

Reason codes for IBM i (ILE RPG)

The reason code parameter (*REASON*) is a qualification to the completion code parameter (*CMPCOD*).

If there is no special reason to report, RCNONE is returned. A successful call returns CCOK and RCNONE.

If the completion code is either CCWARN or CCFAIL, the queue manager always reports a qualifying reason; details are given under each call description.

Where user exit routines set completion codes and reasons, they should adhere to these rules. In addition, any special reason values defined by user exits should be less than zero, to ensure that they do not conflict with values defined by the queue manager. Exits can set reasons already defined by the queue manager, where these are appropriate.

Reason codes also occur in:

- The *DLREA* field of the MQDLH structure
- The *MDFB* field of the MQMD structure

The full list of reason codes is in [API completion and reason codes](#) in .

To find your IBM i reason code in that list, remove the "RC" from the front, for example RC2002 becomes 2002. Also the completion codes there are shown as they are on other platforms:

Table 631.	
IBM i	Other platforms
CCOK	MQCC_OK
CCWARN	MQCC_WARN
CCFAIL	MQCC_FAIL

Rules for validating MQI options for IBM i (ILE RPG)

This topic gives information about the situations that produce an RC2046 reason code from an MQOPEN, MQPUT, MQPUT1, MQGET, or MQCLOSE call.

MQOPEN call on IBM i

For the options of the MQOPEN call:

- *At least one* of the following must be specified:
 - OOBROW
 - OOINPQ
 - OOINPX
 - OOINPS
 - OOINQ

- OOOOUT
- OOSSET
- Only *one* of the following is allowed:
 - OOINPQ
 - OOINPX
 - OOINPS
- Only *one* of the following is allowed:
 - OOBNDQ
 - OOBNDN
 - OOBNDQ

Note: The options listed previously are mutually exclusive. However, because the value of OOBNDQ is zero, specifying it with either of the other two bind options does not result in reason code RC2046. OOBNDQ is provided to aid program documentation.

- If OOSAVA is specified, one of the OOINP* options must also be specified.
- If one of the OOSSET* or OOPAS* options is specified, OOOOUT must also be specified.

MQPUT call on IBM i

For the put-message options:

- The combination of PMSYP and PMNSYP is not allowed.
- Only *one* of the following is allowed:
 - PMDEFC
 - PMNOC
 - PMPASA
 - PMPASI
 - PMSETA
 - PMSETI
- PMALTU is not allowed (it is valid only on the MQPUT1 call).

MQPUT1 call on IBM i

For the put-message options, the rules are the same as for the MQPUT call, except for the following options:

- PMALTU is allowed.
- PMLOGO is *not* allowed.

MQGET call on IBM i

For the get-message options:

- Only *one* of the following options is allowed:
 - GMNSYP
 - GMSYP
 - GMPSYP
- Only *one* of the following options is allowed:
 - GMBRWF

- GMBRWC
- GMBRWN
- GMMUC
- GMSYP is not allowed with any of the following options:
 - GMBRWF
 - GMBRWC
 - GMBRWN
 - GMLK
 - GMUNLK
- GMPSP is not allowed with any of the following options:
 - GMBRWF
 - GMBRWC
 - GMBRWN
 - GMCMPM
 - GMUNLK
- If GMLK is specified, one of the following options must also be specified:
 - GMBRWF
 - GMBRWC
 - GMBRWN
- If GMUNLK is specified, only the following options are allowed:
 - GMNSYP
 - GMNWT

MQCLOSE call on IBM i

- For the options of the MQCLOSE call. The combination of CODEL and COPURG is not allowed.
- Only one of the following is allowed:
 - COKPSB
 - CORMSB

MQSUB call on IBM i

For the options of the MQSUB call:

- At least one of the following must be specified:
- At least one of the following must be specified:
 - SOALT
 - SORES
 - SOCRT
- Only one of the following is allowed:
 - SODUR
 - SONDUR

Note: The options listed previously are mutually exclusive. However, as the value of SONDUR is zero, specifying it with SODUR does not result in reason code RC2046. SONDUR is provided to aid program documentation.

- The combination of SOGRP and SOMAN is not allowed.

- SOGRP requires SOSCID to be specified.
- Only one of the following is allowed: SOAUID SOFUID
- The combination of SONEWP and SOPUBR is not allowed.
- SONEWP is only allowed in combination with SOCRT.
- Only one of the following is allowed:
 - SOWCHR
 - SOWTOP

Machine encodings on IBM i

Use this information to learn about the structure of the *MDENC* field in the message descriptor.

For more information about the message descriptor, see [“MQMD - Message descriptor”](#) on page 1068.

The *MDENC* field is a 32-bit integer that is divided into four separate subfields; these subfields identify:

- The encoding used for binary integers
- The encoding used for packed-decimal integers
- The encoding used for floating-point numbers
- Reserved bits

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined:

ENIMSK

Mask for binary-integer encoding.

This subfield occupies bit positions 28 through 31 within the *MDENC* field.

ENDMSK

Mask for packed-decimal-integer encoding.

This subfield occupies bit positions 24 through 27 within the *MDENC* field.

ENFMSK

Mask for floating-point encoding.

This subfield occupies bit positions 20 through 23 within the *MDENC* field.

ENRMSK

Mask for reserved bits.

This subfield occupies bit positions 0 through 19 within the *MDENC* field.

Binary-integer encoding

Valid values for binary-integer encoding.

The following values are valid for the binary-integer encoding:

ENIUND

Undefined integer encoding.

Binary integers are represented using an encoding that is undefined.

ENINOR

Normal integer encoding.

Binary integers are represented in the conventional way:

- The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address.

- The least significant bit in each byte is next to the byte with the next higher address; the most significant bit in each byte is next to the byte with the next lower address.

ENIREV

Reversed integer encoding.

Binary integers are represented in the same way as ENINOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENINOR.

Packed-decimal-integer encoding

Valid values for packed-decimal-integer encoding

The following values are valid for the packed-decimal-integer encoding:

ENDUND

Undefined packed-decimal encoding.

Packed-decimal integers are represented using an encoding that is undefined.

ENDNOR

Normal packed-decimal encoding.

Packed-decimal integers are represented in the conventional way:

- Each decimal digit in the printable form of the number is represented in packed decimal by a single hexadecimal digit in the range X'0' through X'9'. Each hexadecimal digit occupies 4 bits, and so each byte in the packed decimal number represents two decimal digits in the printable form of the number.
- The least significant byte in the packed-decimal number is the byte which contains the least significant decimal digit. Within that byte, the most significant 4 bits contain the least significant decimal digit, and the least significant 4 bits contain the sign. The sign is either X'C' (positive), X'D' (negative), or X'F' (unsigned).
- The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address.
- The least significant bit in each byte is next to the byte with the next higher address; the most significant bit in each byte is next to the byte with the next lower address.

ENDREV

Reversed packed-decimal encoding.

Packed-decimal integers are represented in the same way as ENDNOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENDNOR.

Floating-point encoding

Valid values for floating-point encoding

The following values are valid for the floating-point encoding:

ENFUND

Undefined floating-point encoding.

Floating-point numbers are represented using an encoding that is undefined.

ENFNOR

Normal IEEE (The Institute of Electrical and Electronics Engineers) float encoding.

Floating-point numbers are represented using the standard IEEE floating-point format, with the bytes arranged as follows:

- The least significant byte in the mantissa has the highest address of any of the bytes in the number; the byte containing the exponent has the lowest address
- The least significant bit in each byte is next to the byte with the next higher address; the most significant bit in each byte is next to the byte with the next lower address

Details of the IEEE float encoding might be found in IEEE Standard 754.

ENFREV

Reversed IEEE float encoding.

Floating-point numbers are represented in the same way as ENFNOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENFNOR.

ENF390

System/390 architecture float encoding.

Floating-point numbers are represented using the standard System/390 floating-point format; this is also used by System/370.

Constructing encodings

To construct a value for the *MDENC* field in MQMD, the relevant constants that describe the required encodings should be added.

Be sure to combine only one of the ENI* encodings with one of the END* encodings and one of the ENF* encodings.

Analyzing encodings

The *MDENC* field contains subfields; because of this, applications that need to examine the integer, packed decimal, or float encoding should use the technique described in this topic.

Using arithmetic

The following steps should be performed using integer arithmetic:

1. Select one of the following values, according to the type of encoding required:

- 1 for the binary integer encoding
- 16 for the packed decimal integer encoding
- 256 for the floating point encoding

Call the value A.

2. Divide the value of the *MDENC* field by A ; call the result B.

3. Divide B by 16; call the result C.

4. Multiply C by 16 and subtract from B ; call the result D.

5. Multiply D by A ; call the result E.

6. E is the encoding required, and can be tested for equality with each of the values that is valid for that type of encoding.

Summary of machine architecture encodings

A table summarizing encodings for machine architectures.

Encodings for machine architectures are shown in [Table 632 on page 1387](#).

<i>Table 632. Summary of encodings for machine architectures</i>			
Machine architecture	Binary integer encoding	Packed-decimal integer encoding	Floating-point encoding
IBM i	normal	normal	IEEE normal
Intel x86	reversed	reversed	IEEE reversed
PowerPC	normal	normal	IEEE normal

Table 632. Summary of encodings for machine architectures (continued)			
Machine architecture	Binary integer encoding	Packed-decimal integer encoding	Floating-point encoding
System/390	normal	normal	System/390

Report options and message flags

This topic concerns the *MDREP* and *MDMFL* fields that are part of the message descriptor MQMD specified on the MQGET, MQPUT, and MQPUT1 calls.

For more information about the message descriptor, see [“MQMD - Message descriptor”](#) on page 1068. This information describes:

- The structure of the report field and how the queue manager processes it
- How an application should analyze the report field
- The structure of the message-flags field

Structure of the report field

The *MDREP* field is a 32-bit integer that is divided into three separate subfields.

These subfields identify:

- Report options that are rejected if the local queue manager does not recognize them
- Report options that are always accepted, even if the local queue manager does not recognize them
- Report options that are accepted only if certain other conditions are satisfied

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. Note that the bits in a subfield are not necessarily adjacent. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

RORUM

Mask for unsupported report options that are rejected.

This mask identifies the bit positions within the *MDREP* field where report options which are not supported by the local queue manager will cause the MQPUT or MQPUT1 call to fail with completion code CCFAIL and reason code RC2061.

This subfield occupies bit positions 3, and 11 through 13.

ROAUM

Mask for unsupported report options that are accepted.

This mask identifies the bit positions within the *MDREP* field where report options which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls. Completion code CCWARN with reason code RC2104 are returned in this case.

This subfield occupies bit positions 0 through 2, 4 through 10, and 24 through 31.

The following report options are included in this subfield:

- ROCMTC
- RODLQ
- RODISC
- ROEXC
- ROEXCD
- ROEXCF
- ROEXP

- ROEXPD
- ROEXPF
- RONAN
- RONMI
- RONONE
- ROPAN
- ROPCI
- ROPMI

ROAUXM

Mask for unsupported report options that are accepted only in certain circumstances.

This mask identifies the bit positions within the *MDREP* field where report options which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ODMN* and *ODON* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code CCWARN with reason code RC2104 are returned if these conditions are satisfied, and CCFAIL with reason code RC2061 if not.

This subfield occupies bit positions 14 through 23.

The following report options are included in this subfield:

- ROCOA
- ROCOAD
- ROCOAF
- ROCOD
- ROCODD
- ROCODF

If there are any options specified in the *MDREP* field which the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *MDREP* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described previously are returned.

If CCWARN is returned, it is not defined which reason code is returned if other warning conditions exist.

The ability to specify and have accepted report options which are not recognized by the local queue manager is useful when it is necessary to send a message with a report option which will be recognized and processed by a *remote* queue manager.

Analyzing the report field

The MDREP field contains subfields. Because of this, some applications need to check whether the sender of the message requested a particular report. Those applications should use the technique described in this topic.

Using arithmetic

The following steps should be performed using integer arithmetic:

1. Select one of the following values, according to the type of report to be checked:
 - ROCOA for COA report

- ROCOD for COD report
- ROEXC for exception report
- ROEXP for expiration report

Call the value A.

2. Divide the *MDREP* field by A ; call the result B.
3. Divide B by 8 ; call the result C.
4. Multiply C by 8 and subtract from B ; call the result D.
5. Multiply D by A ; call the result E.
6. Test E for equality with each of the values that is possible for that type of report.

For example, if A is ROEXC, test E for equality with each of the following to determine what was specified by the sender of the message:

- RONONE
- ROEXC
- ROEXCD
- ROEXCF

The tests can be performed in whatever order is most convenient for the application logic.

The following pseudocode illustrates this technique for exception report messages:

```
A = ROEXC
B = Report/A
C = B/8
D = B - C*8
E = D*A
```

A similar method can be used to test for the ROPMI or ROPCI options; select as the value A whichever of these two constants is appropriate, and then proceed as described previously, but replacing the value 8 in the previous steps by the value 2.

Structure of the message-flags field

The *MDMFL* field is a 32-bit integer that is divided into three separate subfields.

These subfields identify:

- Message flags that are rejected if the local queue manager does not recognize them
- Message flags that are always accepted, even if the local queue manager does not recognize them
- Message flags that are accepted only if certain other conditions are satisfied

Note: All subfields in *MDMFL* are reserved for use by the queue manager.

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MFRUM

Mask for unsupported message flags that are rejected.

This mask identifies the bit positions within the *MDMFL* field where message flags which are not supported by the local queue manager will cause the MQPUT or MQPUT1 call to fail with completion code CCFAIL and reason code RC2249.

This subfield occupies bit positions 20 through 31.

The following message flags are included in this subfield:

- MFLMIG
- MFLSEG

- MFMIG
- MFSEG
- MFSEGA
- MFSEGI

MFAUM

Mask for unsupported message flags that are accepted.

This mask identifies the bit positions within the *MDMFL* field where message flags which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls. The completion code is CCOK.

This subfield occupies bit positions 0 through 11.

MFAUXM

Mask for unsupported message flags that are accepted only in certain circumstances.

This mask identifies the bit positions within the *MDMFL* field where message flags which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ODMN* and *ODON* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code CCOK is returned if these conditions are satisfied, and CCFAIL with reason code RC2249 if not.

This subfield occupies bit positions 12 through 19.

If there are flags specified in the *MDMFL* field that the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *MDMFL* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described previously are returned.

Data conversion

This topic describes the interface to the data-conversion exit, and the processing performed by the queue manager when data conversion is required.

The data-conversion exit is invoked as part of the processing of the MQGET call in order to convert the application message data to the representation required by the receiving application. Conversion of the application message data is optional - it requires the GMCONV option to be specified on the MQGET call.

The following are described:

- The processing performed by the queue manager in response to the GMCONV option; see [“Conversion processing” on page 1392](#).
- Processing conventions used by the queue manager when processing a built-in format; these conventions are recommended for user-written exits too. See [“Processing conventions” on page 1393](#).
- Special considerations for the conversion of report messages; see [“Conversion of report messages” on page 1397](#).
- The parameters passed to the data-conversion exit; see [“MQCONVX - Data conversion exit” on page 1407](#).
- A call that can be used from the exit in order to convert character data between different representations; see [“MQXCNVC - Convert characters” on page 1402](#).
- The data-structure parameter which is specific to the exit; see [“MQDXP - Data-conversion exit parameter” on page 1398](#).

Conversion processing

This information describes the processing performed by the queue manager in response to the GMCONV option.

The queue manager performs the following actions if the GMCONV option is specified on the MQGET call, and there is a message to be returned to the application:

1. If one or more of the following is true, no conversion is necessary:

- The message data is already in the character set and encoding required by the application issuing the MQGET call. The application must set the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter of the MQGET call to the values required, before issuing the call.
- The length of the message data is zero.
- The length of the *BUFFER* parameter of the MQGET call is zero.

In these cases the message is returned without conversion to the application issuing the MQGET call; the *MDCSI* and *MDENC* values in the *MSGDSC* parameter are set to the values in the control information in the message, and the call completes with one of the following combinations of completion code and reason code:

Completion code

Reason code

CCOK

RCNONE

CCWARN

RC2079

CCWARN

RC2080

The following steps are performed only if the character set or encoding of the message data differs from the corresponding value in the *MSGDSC* parameter, and there is data to be converted:

1. If the *MDFMT* field in the control information in the message has the value FMNONE, the message is returned unconverted, with completion code CCWARN and reason code RC2110.

In all other cases conversion processing continues.

2. The message is removed from the queue and placed in a temporary buffer which is the same size as the *BUFFER* parameter. For browse operations, the message is copied into the temporary buffer, instead of being removed from the queue.

3. If the message has to be truncated to fit in the buffer, the following is done:

- If the GMATM option was *not* specified, the message is returned unconverted, with completion code CCWARN and reason code RC2080.
- If the GMATM option was specified, the completion code is set to CCWARN, the reason code is set to RC2079, and conversion processing continues.

4. If the message can be accommodated in the buffer without truncation, or the GMATM option was specified, the following is done:

- If the format is a built-in format, the buffer is passed to the queue manager's data-conversion service.
- If the format is not a built-in format, the buffer is passed to a user-written exit which has the same name as the format. If the exit cannot be found, the message is returned unconverted, with completion code CCWARN and reason code RC2110.

If no error occurs, the output from the data-conversion service or from the user-written exit is the converted message, plus the completion code and reason code to be returned to the application issuing the MQGET call.

5. If the conversion is successful, the queue manager returns the converted message to the application. In this case, the completion code and reason code returned by the MQGET call will typically be one of the following combinations:

Completion code

Reason code

CCOK

RCNONE

CCWARN

RC2079

However, if the conversion is performed by a user-written exit, other reason codes can be returned, even when the conversion is successful.

If the conversion fails (for whatever reason), the queue manager returns the unconverted message to the application, with the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter set to the values in the control information in the message, and with completion code CCWARN.

Processing conventions

When converting a built-in format, the queue manager follows the processing conventions described in this topic.

Consider applying these conventions to user-written exits, although this is not enforced by the queue manager. The built-in formats converted by the queue manager are:

Built in formats
FMADMN
FMMDE
FMCICS
FMPCF
FMCMDB1
FMRMH
FMCMDB2
FMRFH
FMDLH
FMRFH2
FMDH
FMSTR
FMEVNT
FMTM
FMIMS
FMXQH
FMIMVS

1. If the message expands during conversion, and exceeds the size of the *BUFFER* parameter, the following is done:
 - If the GMATM option was *not* specified, the message is returned unconverted, with completion code CCWARN and reason code RC2120.

- If the GMATM option was specified, the message is truncated, the completion code is set to CCWARN, the reason code is set to RC2079, and conversion processing continues.
2. If truncation occurs (either before or during conversion), it is possible for the number of valid bytes returned in the *BUFFER* parameter to be *less than* the length of the buffer.

This can occur, for example, if a 4-byte integer or a DBCS character straddles the end of the buffer. The incomplete element of information is not converted, and so those bytes in the returned message do not contain valid information. This can also occur if a message that was truncated before conversion shrinks during conversion.

If the number of valid bytes returned is less than the length of the buffer, the unused bytes at the end of the buffer are set to nulls.
 3. If an array or string straddles the end of the buffer, as much of the data as possible is converted; only the particular array element or DBCS character which is incomplete is not converted - preceding array elements or characters are converted.
 4. If truncation occurs (either before or during conversion), the length returned for the *DATLEN* parameter is the length of the *unconverted* message before truncation.
 5. When strings are converted between single-byte character sets (SBCS), double-byte character sets (DBCS), or multi-byte character sets (MBCS), the strings can expand or contract.
 - In the PCF formats FMADMN, FMEVNT, and FMPCF, the strings in the MQCFST and MQCFSL structures expand or contract as necessary to accommodate the string after conversion.

For the string-list structure MQCFSL, the strings in the list might expand or contract by different amounts. If this happens, the queue manager pads the shorter strings with blanks to make them the same length as the longest string after conversion.
 - In the format FMRMH, the strings addressed by the *RMSEO*, *RMSNO*, *RMDEO*, and *RMDNO* fields expand or contract as necessary to accommodate the strings after conversion.
 - In the format FMRFH, the *RFNVS* field expands or contracts as necessary to accommodate the name/value pairs after conversion.
 - In structures with fixed field sizes, the queue manager allows strings to expand or contract within their fixed fields, if no significant information is lost. In this regard, trailing blanks and characters following the first null character in the field are treated as insignificant.
 - If the string expands, but only insignificant characters need to be discarded to accommodate the converted string in the field, the conversion succeeds and the call completes with CCOK and reason code RCNONE (assuming no other errors).
 - If the string expands, but the converted string requires significant characters to be discarded in order to fit in the field, the message is returned unconverted and the call completes with CCWARN and reason code RC2190.

Note: Reason code RC2190 results in this case whether the GMATM option was specified.

 - If the string contracts, the queue manager pads the string with blanks to the length of the field.
 6. For messages consisting of one or more IBM MQ header structures followed by user data, it is possible for one or more of the header structures to be converted, while the remainder of the message is not. However, with two exceptions, the *MDCSI* and *MDENC* fields in each header structure always correctly indicate the character set and encoding of the data that follows the header structure.

The two exceptions are the MQCIH and MQIIH structures, where the values in the *MDCSI* and *MDENC* fields in those structures are not significant. For those structures, the data following the structure is in the same character set and encoding as the MQCIH or MQIIH structure itself.
 7. If the *MDCSI* or *MDENC* fields in the control information of the message being retrieved, or in the *MSGDSC* parameter, specify values which are undefined or not supported, the queue manager might ignore the error if the undefined or unsupported value does not need to be used in converting the message.

For example, if the *MDENC* field in the message specifies an unsupported float encoding, but the message contains only integer data, or contains floating-point data which does not require

conversion (because the source and target float encodings are identical), the error might or might not be diagnosed.

If the error is diagnosed, the message is returned unconverted, with completion code CCWARN and one of the RC2111, RC2112, RC2113, RC2114 or RC2115, RC2116, RC2117, RC2118 reason codes (as appropriate); the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to the values in the control information in the message.

If the error is not diagnosed and the conversion completes successfully, the values returned in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter, are those specified by the application issuing the MQGET call.

8. In all cases, if the message is returned to the application unconverted the completion code is set to CCWARN, and the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to the values appropriate to the unconverted data. This is done for FMNONE also.

The *REASON* parameter is set to a code that indicates why the conversion could not be carried out, unless the message also had to be truncated; reason codes related to truncation take precedence over reason codes related to conversion. (To determine if a truncated message was converted, check the values returned in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter.)

When an error is diagnosed, either a specific reason code is returned, or the general reason code RC2119. The reason code returned depends on the diagnostic capabilities of the underlying data-conversion service.

9. If completion code CCWARN is returned, and more than one reason code is relevant, the order of precedence is as follows:

- a. The following reason takes precedence over all others:
 - RC2079
- b. Next in precedence is the following reason:
 - RC2110
- c. The order of precedence within the remaining reason codes is not defined.

10. On completion of the MQGET call:

- The following reason code indicates that the message was converted successfully:
 - RCNONE
- The following reason code indicates that the message *may* have been converted successfully (check the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to find out):
 - RC2079
- All other reason codes indicate that the message was not converted.

The following processing is specific to the built-in formats; it is not applicable to user-defined formats:

1. Except for the following formats:

- FMADMN
- FMEVNT
- FMIMVS
- FMPCF
- FMSTR

none of the built-in formats can be converted from or to character sets that do not have SBCS characters for the characters that are valid in queue names. If an attempt is made to perform such a conversion, the message is returned unconverted, with completion code CCWARN and reason code RC2111 or RC2115, as appropriate.

The Unicode character set UCS-2 is an example of a character set that does not have SBCS characters for the characters that are valid in queue names.

2. If the message data for a built-in format is truncated, fields within the message which contain lengths of strings, or counts of elements or structures, are *not* adjusted to reflect the length of the data returned to the application; the values returned for such fields within the message data are the values applicable to the message *before truncation*.

When processing messages such as a truncated FMADMN message, care must be taken to ensure that the application does not attempt to access data beyond the end of the data returned.

3. If the format name is FMDLH, the message data begins with an MQDLH structure, and this may be followed by zero or more bytes of application message data. The format, character set, and encoding of the application message data are defined by the *DLFMT*, *DLCSI*, and *DLENC* fields in the MQDLH structure at the start of the message. Since the MQDLH structure and application message data can have different character sets and encodings, it is possible for one, other, or both of the MQDLH structure and application message data to require conversion.

The queue manager converts the MQDLH structure first, as necessary. If conversion is successful, or the MQDLH structure does not require conversion, the queue manager checks the *DLCSI* and *DLENC* fields in the MQDLH structure to see if conversion of the application message data is required. If conversion is required, the queue manager invokes the user-written exit with the name given by the *DLFMT* field in the MQDLH structure, or performs the conversion itself (if *DLFMT* is the name of a built-in format).

If the MQGET call returns a completion code of CCWARN, and the reason code is one of those indicating that conversion was not successful, one of the following applies:

- The MQDLH structure could not be converted. In this case the application message data will not have been converted either.
- The MQDLH structure was converted, but the application message data was not.

The application can examine the values returned in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter, and those in the MQDLH structure, in order to determine which of the previous applies.

4. If the format name is FMXQH, the message data begins with an MQXQH structure, and this may be followed by zero or more bytes of additional data. This additional data is typically the application message data (which may be of zero length), but there can also be one or more further IBM MQ header structures present, at the start of the additional data.

The MQXQH structure must be in the character set and encoding of the queue manager. The format, character set, and encoding of the data following the MQXQH structure are given by the *MDFMT*, *MDCSI*, and *MDENC* fields in the MQMD structure contained *within* the MQXQH. For each subsequent IBM MQ header structure present, the *MDFMT*, *MDCSI*, and *MDENC* fields in the structure describe the data that follows that structure; that data is either another IBM MQ header structure, or the application message data.

If the GMCONV option is specified for an FMXQH message, the application message data and certain of the MQ header structures are converted, *but the data in the MQXQH structure is not*. On return from the MQGET call, therefore:

- The values of the *MDFMT*, *MDCSI*, and *MDENC* fields in the *MSGDSC* parameter, describe the data in the MQXQH structure, and *not* the application message data; the values will therefore *not* be the same as those specified by the application that issued the MQGET call.

The effect of this is that an application which repeatedly gets messages from a transmission queue with the GMCONV option specified must reset the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to the values necessary for the application message data, before each MQGET call.

- The values of the *MDFMT*, *MDCSI*, and *MDENC* fields in the last MQ header structure present describe the application message data. If there are no other IBM MQ header structures present, the application message data is described by these fields in the MQMD structure within the MQXQH structure. If conversion is successful, the values will be the same as those specified in the *MSGDSC* parameter by the application that issued the MQGET call.

If the message is a distribution-list message, the MQXQH structure is followed by an MQDH structure (plus its arrays of MQOR and MQPMR records), which in turn may be followed by zero or more further

IBM MQ header structures and zero or more bytes of application message data. Like the MQXQH structure, the MQDH structure must be in the character set and encoding of the queue manager, and it is not converted on the MQGET call, even if the GMCONV option is specified.

The processing of the MQXQH and MQDH structures described previously are primarily intended for use by message channel agents when they get messages from transmission queues.

Conversion of report messages

A report message can contain varying amounts of application message data, according to the report options specified by the sender of the original message.

In particular, a report message can contain either:

1. No application message data
2. Some of the application message data from the original message

This occurs when the sender of the original message specifies RO*D and the message is longer than 100 bytes.

3. All of the application message data from the original message

This occurs when the sender of the original message specifies RO*F, or specifies RO*D and the message is 100 bytes or shorter.

When the queue manager or message channel agent generates a report message, it copies the format name from the original message into the *MDFMT* field in the control information in the report message. The format name in the report message might therefore imply a length of data which is different from the length present in the report message (cases 1 and 2 described previously).

If the GMCONV option is specified when the report message is retrieved:

- For case 1 described previously, the data-conversion exit will not be invoked (because the report message will have no data).
- For case 3 described previously, the format name correctly implies the length of the message data.
- But for case 2 described previously, the data-conversion exit will be invoked to convert a message which is *shorter* than the length implied by the format name.

In addition, the reason code passed to the exit will typically be RCNONE (that is, the reason code will not indicate that the message has been truncated). This happens because the message data was truncated by the *sender* of the report message, and not by the receiver's queue manager in response to the MQGET call.

Because of these possibilities, the data-conversion exit should *not* use the format name to deduce the length of data passed to it; instead the exit should check the length of data provided, and be prepared to convert *less* data than the length implied by the format name. If the data can be converted successfully, completion code CCOK and reason code RCNONE should be returned by the exit. The length of the message data to be converted is passed to the exit as the *INLEN* parameter.

Product-sensitive programming interface

If a report message contains information about an activity that has taken place, it is known as an activity report. Examples of activities are:

- an MCA sending a message from a queue down a channel
- an MCA receiving a message from a channel and putting it onto a queue
- an MCA dead-letter queuing an undeliverable message
- an MCA getting a message off a queue and discarding it
- a dead-letter handler placing a message back on a queue
- the command server processing a PCF request - a broker processing a publish request
- a user application getting a message from a queue - a user application browsing a message on a queue

Any application, including the queue manager, can add some of the message data to the activity report following the report header. The amount of data that should be supplied if some is sent is not fixed, and is decided by the application. The information returned should be useful to the application processing the activity report. Queue manager activity reports will return with them any standard IBM MQ header structures (beginning 'MQH') contained in the original message. This includes, for example, any MQRFH2 headers that were included in the original message. Also the queue manager will return an MQCFH header found, but not the PCF parameters associated with it. This gives monitoring applications an idea of what the message was about.

MQDXP - Data-conversion exit parameter

Data-conversion exit parameter block.

Overview

Purpose: The MQDXP structure is a parameter that the queue manager passes to the data-conversion exit when the exit is invoked to convert the message data as part of the processing of the MQGET call. See the description of the MQCONVX call for details of the data conversion exit.

Character set and encoding: Character data in MQDXP is in the character set of the local queue manager; this is given by the *CodedCharSetId* queue manager attribute. Numeric data in MQDXP is in the native machine encoding; this is given by ENNAT.

Usage: Only the *DXLEN*, *DXCC*, *DXREA* and *DXRES* fields in MQDXP might be changed by the exit; changes to other fields are ignored. However, the *DXLEN* field *cannot* be changed if the message being converted is a segment that contains only part of a logical message.

When control returns to the queue manager from the exit, the queue manager checks the values returned in MQDXP. If the values returned are not valid, the queue manager continues processing as though the exit had returned XRFAIL in *DXRES* ; however, the queue manager ignores the values of the *DXCC* and *DXREA* fields returned by the exit in this case, and uses instead the values those fields had on *input* to the exit. The following values in MQDXP cause this processing to occur:

- *DXRES* field not XROK and not XRFAIL
- *DXCC* field not CCOK and not CCWARN
- *DXLEN* field less than zero, or *DXLEN* field changed when the message being converted is a segment that contains only part of a logical message.
- [“Fields” on page 1398](#)
- [“RPG declaration \(copy file CMQDXPH\)” on page 1402](#)

Fields

The MQDXP structure contains the following fields; the fields are described in **alphabetical order**:

DXAOP (10-digit signed integer)

Application options.

This is a copy of the *GMOPT* field of the MQGMO structure specified by the application issuing the MQGET call. The exit may need to examine these to ascertain whether the GMATM option was specified.

This is an input field to the exit.

DXCC (10-digit signed integer)

Completion code.

When the exit is invoked, this contains the completion code that will be returned to the application that issued the MQGET call, if the exit chooses to do nothing. It is always CCWARN, because either the message was truncated, or the message requires conversion and this has not yet been done.

On output from the exit, this field contains the completion code to be returned to the application in the *CMPCOD* parameter of the MQGET call; only CCOK and CCWARN are valid. See the description of the *DXREA* field for suggestions on how the exit should set this field on output.

This is an input/output field to the exit.

DXCSI (10-digit signed integer)

Character set required by application.

This is the coded character-set identifier of the character set required by the application issuing the MQGET call; see the *MDCSI* field in the MQMD structure for more details. If the application specifies the special value CSQM on the MQGET call, the queue manager changes this to the actual character-set identifier of the character set used by the queue manager, before invoking the exit.

If the conversion is successful, the exit should copy this to the *MDCSI* field in the message descriptor.

This is an input field to the exit.

DXENC (10-digit signed integer)

Numeric encoding required by application.

This is the numeric encoding required by the application issuing the MQGET call; see the *MDENC* field in the MQMD structure for more details.

If the conversion is successful, the exit should copy this to the *MDENC* field in the message descriptor.

This is an input field to the exit.

DXHCN (10-digit signed integer)

Connection handle.

This is a connection handle which can be used on the MQXCNVC call. This handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

DXLEN (10-digit signed integer)

Length in bytes of message data.

When the exit is invoked, this field contains the original length of the application message data. If the message was truncated in order to fit into the buffer provided by the application, the size of the message provided to the exit will be *smaller* than the value of *DXLEN*. The size of the message provided to the exit is always given by the *INLEN* parameter of the exit, irrespective of any truncation that may have occurred.

Truncation is indicated by the *DXREA* field having the value RC2079 on input to the exit.

Most conversions will not need to change this length, but an exit can do so if necessary; the value set by the exit is returned to the application in the *DATLEN* parameter of the MQGET call. However, this length *cannot* be changed if the message being converted is a segment that contains only part of a logical message. This is because changing the length would cause the offsets of later segments in the logical message to be incorrect.

Note that, if the exit wants to change the length of the data, be aware that the queue manager has already decided whether the message data will fit into the application's buffer, based on the length of the *unconverted* data. This decision determines whether the message is removed from the queue (or the browse cursor moved, for a browse request), and is not affected by any change to the data length caused by the conversion. For this reason it is recommended that conversion exits do not cause a change in the length of the application message data.

If character conversion does imply a change of length, a string can be converted into another string with the same length in bytes, truncating trailing blanks, or padding with blanks as necessary.

The exit is not invoked if the message contains no application message data; hence *DXLEN* is always greater than zero.

This is an input/output field to the exit.

DXREA (10-digit signed integer)

Reason code qualifying *DXCC*.

When the exit is invoked, this contains the reason code that will be returned to the application that issued the MQGET call, if the exit chooses to do nothing. Among possible values are RC2079, indicating that the message was truncated in order to fit into the buffer provided by the application, and RC2119, indicating that the message requires conversion but that this has not yet been done.

On output from the exit, this field contains the reason to be returned to the application in the *REASON* parameter of the MQGET call; the following is recommended:

- If *DXREA* had the value RC2079 on input to the exit, the *DXREA* and *DXCC* fields should not be altered, irrespective of whether the conversion succeeds or fails.

(If the *DXCC* field is not CCOK, the application which retrieves the message can identify a conversion failure by comparing the returned *MDENC* and *MDCSI* values in the message descriptor with the values requested; in contrast, the application cannot distinguish a truncated message from a message that just fitted the buffer. For this reason, RC2079 should be returned in preference to any of the reasons that indicate conversion failure.)

- If *DXREA* had any other value on input to the exit:
 - If the conversion succeeds, *DXCC* should be set to CCOK and *DXREA* set to RCNONE.
 - If the conversion fails, or the message expands and has to be truncated to fit in the buffer, *DXCC* should be set to CCWARN (or left unchanged), and *DXREA* set to one of the values in the following list, to indicate the nature of the failure.

Note that, if the message after conversion is too big for the buffer, it should be truncated only if the application that issued the MQGET call specified the GMATM option:

- If it did specify that option, reason RC2079 should be returned.
- If it did not specify that option, the message should be returned unconverted, with reason code RC2120.

The reason codes in the following list are recommended for use by the exit to indicate the reason that conversion failed, but the exit can return other values from the set of RC* codes if deemed appropriate. In addition, the range of values RC0900 through RC0999 are allocated for use by the exit to indicate conditions that the exit wants to communicate to the application issuing the MQGET call.

Note: If the message cannot be converted successfully, the exit *must* return XRFAIL in the *DXRES* field, in order to cause the queue manager to return the unconverted message. This is true regardless of the reason code returned in the *DXREA* field.

RC0900

(900, X'384') Lowest value for application-defined reason code.

RC0999

(999, X'3E7') Highest value for application-defined reason code.

RC2120

(2120, X'848') Converted data too big for buffer.

RC2119

(2119, X'847') Message data not converted.

RC2111

(2111, X'83F') Source coded character set identifier not valid.

RC2113

(2113, X'841') Packed-decimal encoding in message not recognized.

RC2114

(2114, X'842') Floating-point encoding in message not recognized.

RC2112

(2112, X'840') Source integer encoding not recognized.

RC2115

(2115, X'843') Target coded character set identifier not valid.

RC2117

(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

RC2118

(2118, X'846') Floating-point encoding specified by receiver not recognized.

RC2116

(2116, X'844') Target integer encoding not recognized.

RC2079

(2079, X'81F') Truncated message returned (processing completed).

This is an input/output field to the exit.

DXRES (10-digit signed integer)

Response from exit.

This is set by the exit to indicate the success or otherwise of the conversion. It must be one of the following:

XROK

Conversion was successful.

If the exit specifies this value, the queue manager returns the following to the application that issued the MQGET call:

- The value of the *DXCC* field on output from the exit
- The value of the *DXREA* field on output from the exit
- The value of the *DXLEN* field on output from the exit
- The contents of the exit's output buffer *OUTBUF*. The number of bytes returned is the lesser of the exit's *OUTLEN* parameter, and the value of the *DXLEN* field on output from the exit

If the *MDENC* and *MDCSI* fields in the exit's message descriptor parameter are *both* unchanged, the queue manager returns:

- The value of the *MDENC* and *MDCSI* fields in the MQDXP structure on *input* to the exit

If one or both of the *MDENC* and *MDCSI* fields in the exit's message descriptor parameter has been changed, the queue manager returns:

- The value of the *MDENC* and *MDCSI* fields in the exit's message descriptor parameter on output from the exit
-

XRFAIL

Conversion was unsuccessful.

If the exit specifies this value, the queue manager returns the following to the application that issued the MQGET call:

- The value of the *DXCC* field on output from the exit
- The value of the *DXREA* field on output from the exit
- The value of the *DXLEN* field on *input* to the exit
- The contents of the exit's input buffer *INBUF*. The number of bytes returned is given by the *INLEN* parameter

If the exit has altered *INBUF*, the results are undefined.

DXRES is an output field from the exit.

DXSID (4-byte character string)

Structure identifier.

The value must be:

DXSIDV

Identifier for data conversion exit parameter structure.

This is an input field to the exit.

DXVER (10-digit signed integer)

Structure version number.

The value must be:

DXVER1

Version number for data-conversion exit parameter structure.

The following constant specifies the version number of the current version:

DXVERC

Current version of data-conversion exit parameter structure.

Note: When a new version of this structure is introduced, the layout of the existing part is not changed. The exit should therefore check that the *DXVER* field is equal to or greater than the lowest version which contains the fields that the exit needs to use.

This is an input field to the exit.

DXXOP (10-digit signed integer)

Reserved.

This is a reserved field; its value is 0.

RPG declaration (copy file CMQDXPH)

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQDXP Structure
D*
D* Structure identifier
D DXSID          1          4
D* Structure version number
D DXVER          5          8I 0
D* Reserved
D DXXOP          9          12I 0
D* Application options
D DXAOP         13          16I 0
D* Numeric encoding required by application
D DXENC         17          20I 0
D* Character set required by application
D DXCSI         21          24I 0
D* Length in bytes of message data
D DXLEN         25          28I 0
D* Completion code
D DXCC          29          32I 0
D* Reason code qualifying DXCC
D DXREA         33          36I 0
D* Response from exit
D DXRES         37          40I 0
D* Connection handle
D DXHCN         41          44I 0
```

MQXCNCV - Convert characters

The MQXCNCV call converts characters from one character set to another.

This call is part of the IBM MQ Data Conversion Interface (DCI), which is one of the IBM MQ framework interfaces. Note: This call can be used only from a data-conversion exit.

- [“Syntax” on page 1403](#)
- [“Parameters” on page 1403](#)
- [“RPG invocation \(ILE\)” on page 1407](#)

Syntax

MQXCNVC (*HCONN, OPTS, SRCCSI, SRCLEN, SRCBUF, TGTCSI, TGTLEN, TGTBUF, DATLEN, CMPCOD, REASON*)

Parameters

The MQXCNVC call has the following parameters:

HCONN (10-digit signed integer) - input

Connection handle.

This handle represents the connection to the queue manager. It should normally be the handle passed to the data-conversion exit in the *DXHCN* field of the MQDXP structure; this handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

On IBM i, the following special value can be specified for *HCONN*:

HCDEFH

Default connection handle.

OPTS (10-digit signed integer) - input

Options that control the action of MQXCNVC.

Zero or more of the options described later in this section can be specified. If more than one is required, the values can be added (do not add the same constant more than once).

Default-conversion option: The following option controls the use of default character conversion:

DCCDEF

Default conversion.

This option specifies that default character conversion can be used if one or both of the character sets specified on the call is not supported. This allows the queue manager to use an installation-specified default character set that approximates the specified character set, when converting the string.

Note: The result of using an approximate character set to convert the string is that some characters may be converted incorrectly. This can be avoided by using in the string only characters which are common to both the specified character set and the default character set.

The default character sets are defined by a configuration option when the queue manager is installed or restarted.

If DCCDEF is not specified, the queue manager uses only the specified character sets to convert the string, and the call fails if one or both of the character sets is not supported.

Padding option: The following option allows the queue manager to pad the converted string with blanks or discard insignificant trailing characters, in order to make the converted string fit the target buffer:

DCCFIL

Fill target buffer.

This option requests that conversion take place in such a way that the target buffer is filled completely:

- If the string contracts when it is converted, trailing blanks are added in order to fill the target buffer.
- If the string expands when it is converted, trailing characters that are not significant are discarded to make the converted string fit the target buffer. If this can be done successfully, the call completes with CCOK and reason code RCNONE.

If there are too few insignificant trailing characters, as much of the string as will fit is placed in the target buffer, and the call completes with CCWARN and reason code RC2120.

Insignificant characters are:

- Trailing blanks
- Characters following the first null character in the string (but excluding the first null character itself)
- If the string, *TGTCSI*, and *TGTLEN* are such that the target buffer cannot be set completely with valid characters, the call fails with CCFAIL and reason code RC2144. This can occur when *TGTCSI* is a pure DBCS character set (such as UCS-2), but *TGTLEN* specifies a length that is an odd number of bytes.
- *TGTLEN* can be less than or greater than *SRCLLEN*. On return from MQXCNCV, *DATLEN* has the same value as *TGTLEN*.

If this option is not specified:

- The string is allowed to contract or expand within the target buffer as required. Insignificant trailing characters are not added or discarded.

If the converted string fits in the target buffer, the call completes with CCOK and reason code RCNONE.

If the converted string is too large for the target buffer, as much of the string as will fit is placed in the target buffer, and the call completes with CCWARN and reason code RC2120. Note that fewer than *TGTLEN* bytes can be returned in this case.

- *TGTLEN* can be less than or greater than *SRCLLEN*. On return from MQXCNCV, *DATLEN* is less than or equal to *TGTLEN*.

Encoding options: The following options can be used to specify the integer encodings of the source and target strings. The relevant encoding is used *only* when the corresponding character set identifier indicates that the representation of the character set in main storage is dependent on the encoding used for binary integers. This affects only certain multibyte character sets (for example, UCS-2 character sets).

The encoding is ignored if the character set is a single-byte character set (SBCS), or a multibyte character set with representation in main storage that is not dependent on the integer encoding.

Only one of the DCCS* values should be specified, combined with one of the DCCT* values:

DCCSNA

Source encoding is the default for the environment and programming language.

DCCSNO

Source encoding is normal.

DCCSRE

Source encoding is reversed.

DCCSUN

Source encoding is undefined.

DCCTNA

Target encoding is the default for the environment and programming language.

DCCTNO

Target encoding is normal.

DCCTRE

Target encoding is reversed.

DCCTUN

Target encoding is undefined.

The encoding values defined previously can be added directly to the *OPTS* field. However, if the source or target encoding is obtained from the *MDENC* field in the MQMD or other structure, the following processing must be done:

1. The integer encoding must be extracted from the *MDENC* field by eliminating the float and packed-decimal encodings; see [“Analyzing encodings” on page 1387](#) for details of how to do this.
2. The integer encoding resulting from step 1 must be multiplied by the appropriate factor before being added to the *OPTS* field. These factors are:

DCCSFA

Factor for source encoding

DCCTFA

Factor for target encoding

If not specified, the encoding options default to undefined (DCC*UN). In most cases, this does not affect the successful completion of the MQXCNVC call. However, if the corresponding character set is a multibyte character set with representation that is dependent on the encoding (for example, a UCS-2 character set), the call fails with reason code RC2112 or RC2116 as appropriate.

Default option: If none of the options described previously is specified, the following option can be used:

DCCNON

No options specified.

DCCNON is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

SRCCSI (10-digit signed integer) - input

Coded character set identifier of string before conversion.

This is the coded character set identifier of the input string in *SRCBUF*.

SRCLLEN (10-digit signed integer) - input

Length of string before conversion.

This is the length in bytes of the input string in *SRCBUF* ; it must be zero or greater.

SRCBUF (1-byte character string x SRCLLEN) - input

String to be converted.

This is the buffer containing the string to be converted from one character set to another.

TGTCSI (10-digit signed integer) - input

Coded character set identifier of string after conversion.

This is the coded character set identifier of the character set to which *SRCBUF* is to be converted.

TGTLEN (10-digit signed integer) - input

Length of output buffer.

This is the length in bytes of the output buffer *TGTBUF* ; it must be zero or greater. It can be less than or greater than *SRCLLEN*.

TGTBUF (1-byte character string x TGTLEN) - output

String after conversion.

This is the string after it has been converted to the character set defined by *TGTCSI*. The converted string can be shorter or longer than the unconverted string. The *DATLEN* parameter indicates the number of valid bytes returned.

DATLEN (10-digit signed integer) - output

Length of output string.

This is the length of the string returned in the output buffer *TGTBUF*. The converted string can be shorter or longer than the unconverted string.

CMPCOD (10-digit signed integer) - output

Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) - output

Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2120

(2120, X'848') Converted data too large for buffer.

If *CMPCOD* is CCFAIL:

RC2010

(2010, X'7DA') Data length parameter not valid.

RC2150

(2150, X'866') DBCS string not valid.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2102

(2102, X'836') Insufficient system resources available.

RC2145

(2145, X'861') Source buffer parameter not valid.

RC2111

(2111, X'83F') Source coded character set identifier not valid.

RC2112

(2112, X'840') Source integer encoding not recognized.

RC2143

(2143, X'85F') Source length parameter not valid.

RC2071

(2071, X'817') Insufficient storage available.

RC2146

(2146, X'862') Target buffer parameter not valid.

RC2115

(2115, X'843') Target coded character set identifier not valid.

RC2116

(2116, X'844') Target integer encoding not recognized.

RC2144

(2144, X'860') Target length parameter not valid.

RC2195

(2195, X'893') Unexpected error occurred.

For more information about these reason codes, see [“Return codes for IBM i \(ILE RPG\)” on page 1381](#).

RPG invocation (ILE)

```

C*..1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQXCNCV(HCONN : OPTS : SRCCSI :
C                                SRCLEN : SRCBUF : TGTCSI :
C                                TGTLEN : TGTBUF : DATLEN :
C                                CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQXCNCV      PR      EXTPROC('MQXCNCV')
D* Connection handle
D HCONN              10I 0 VALUE
D* Options that control the action of MQXCNCV
D OPTS              10I 0 VALUE
D* Coded character set identifier of string before conversion
D SRCCSI            10I 0 VALUE
D* Length of string before conversion
D SRCLEN            10I 0 VALUE
D* String to be converted
D SRCBUF              *   VALUE
D* Coded character set identifier of string after conversion
D TGTCSI            10I 0 VALUE
D* Length of output buffer
D TGTLEN            10I 0 VALUE
D* String after conversion
D TGTBUF              *   VALUE
D* Length of output string
D DATLEN            10I 0
D* Completion code
D CMPCOD            10I 0
D* Reason code qualifying CMPCOD
D REASON            10I 0

```

MQCONVX - Data conversion exit

This call definition describes the parameters that are passed to the data-conversion exit.

No entry point called MQCONVX is provided by the queue manager (see usage note [“11” on page 1409](#)).

This definition is part of the IBM MQ Data Conversion Interface (DCI), which is one of the IBM MQ framework interfaces.

- [“Syntax” on page 1408](#)
- [“Usage notes” on page 1408](#)
- [“Parameters” on page 1409](#)
- [“RPG invocation \(ILE\)” on page 1410](#)

Syntax

MQCONVX (MQDXP, MQMD, INLEN, INBUF, OUTLEN, OUTBUF)

Usage notes

1. A data-conversion exit is a user-written exit which receives control during the processing of an MQGET call. The function performed by the data-conversion exit is defined by the provider of the exit; however, the exit must conform to the rules described here, and in the associated parameter structure MQDXP.

The programming languages that can be used for a data-conversion exit are determined by the environment.

2. The exit is invoked only if *all* of the following are true:
 - The GMCONV option is specified on the MQGET call
 - The *MDFMT* field in the message descriptor is not FMNONE
 - The message is not already in the required representation; that is, one or both of the message's *MDCSI* and *MDENC* is different from the value specified by the application in the message descriptor supplied on the MQGET call
 - The queue manager has not already done the conversion successfully
 - The length of the application's buffer is greater than zero
 - The length of the message data is greater than zero
 - The reason code so far during the MQGET operation is RCNONE or RC2079

3. When an exit is being written, consideration should be given to coding the exit in a way that will allow it to convert messages that have been truncated. Truncated messages can arise in the following ways:

- The receiving application provides a buffer that is smaller than the message, but specifies the GMATM option on the MQGET call.

In this case, the *DXREA* field in the *MQDXP* parameter on input to the exit will have the value RC2079.

- The sender of the message truncated it before sending it. This can happen with report messages, for example (see [“Conversion of report messages” on page 1397](#) for more details).

In this case, the *DXREA* field in the *MQDXP* parameter on input to the exit will have the value RCNONE (if the receiving application provided a buffer that was large enough for the message).

Thus the value of the *DXREA* field on input to the exit cannot always be used to decide whether the message has been truncated.

The distinguishing characteristic of a truncated message is that the length provided to the exit in the *INLEN* parameter will be *less than* the length implied by the format name contained in the *MDFMT* field in the message descriptor. The exit should therefore check the value of *INLEN* before attempting to convert any of the data; the exit *should not* assume that the full amount of data implied by the format name has been provided.

If the exit has *not* been written to convert truncated messages, and *INLEN* is less than the value expected, the exit should return XRFAIL in the *DXRES* field of the *MQDXP* parameter, with the *DXCC* field set to CCWARN and the *DXREA* field set to RC2110.

If the exit *has* been written to convert truncated messages, the exit should convert as much of the data as possible (see next usage note), taking care not to attempt to examine or convert data beyond the end of *INBUF*. If the conversion completes successfully, the exit should leave the *DXREA* field in the *MQDXP* parameter unchanged. This returns RC2079 if the message was truncated by the receiver's queue manager, and RCNONE if the message was truncated by the sender of the message.

It is also possible for a message to expand *during* conversion, to the point where it is bigger than *OUTBUF*. In this case the exit must decide whether to truncate the message; the *DXAOP* field in the *MQDXP* parameter will indicate whether the receiving application specified the GMATM option.

4. Generally it is recommended that all of the data in the message provided to the exit in *INBUF* is converted, or that none of it is. An exception to this, however, occurs if the message is truncated, either before conversion or during conversion; in this case there may be an incomplete item at the end of the buffer (for example: one byte of a double-byte character, or 3 bytes of a 4-byte integer). In this situation it is recommended that the incomplete item should be omitted, and unused bytes in *OUTBUF* set to nulls. However, complete elements or characters within an array or string *should* be converted.
5. When an exit is needed for the first time, the queue manager attempts to load an object that has the same name as the format (apart from extensions). The object loaded must contain the exit that processes messages with that format name. It is recommended that the exit name, and the name of the object that contain the exit, should be identical, although not all environments require this.
6. A new copy of the exit is loaded when an application attempts to retrieve the first message that uses that *MDFMT* since the application connected to the queue manager. A new copy might also be loaded at other times, if the queue manager has discarded a previously loaded copy. For this reason, an exit should not attempt to use static storage to communicate information from one invocation of the exit to the next - the exit may be unloaded between the two invocations.
7. If there is a user-supplied exit with the same name as one of the built-in formats supported by the queue manager, the user-supplied exit does not replace the built-in conversion routine. The only circumstances in which such an exit is invoked are:
 - If the built-in conversion routine cannot handle conversions to or from either the *MDCSI* or *MDENC* involved, or
 - If the built-in conversion routine has failed to convert the data (for example, because there is a field or character which cannot be converted).
8. The scope of the exit is environment-dependent. *MDFMT* names should be chosen so as to minimize the risk of clashes with other formats. It is recommended that they start with characters that identify the application defining the format name.
9. The data-conversion exit runs in an environment like that of the program which issued the *MQGET* call; environment includes address space and user profile (where applicable). The program could be a message channel agent sending messages to a destination queue manager that does not support message conversion. The exit cannot compromise the queue manager's integrity, since it does not run in the queue manager's environment.
10. The only *MQI* call which can be used by the exit is *MQXCNVC*; attempting to use other *MQI* calls fails with reason code RC2219, or other unpredictable errors.
11. No entry point called *MQCONVX* is provided by the queue manager. The name of the exit should be the same as the format name (the name contained in the *MDFMT* field in *MQMD*), although this is not required in all environments.

Parameters

The *MQCONVX* call has the following parameters:

MQDXP (MQDXP) - input/output

Data-conversion exit parameter block.

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate the outcome of the conversion. See [“MQDXP - Data-conversion exit parameter”](#) on page 1398 for details of the fields in this structure.

MQMD (MQMD) - input/output

Message descriptor.

On input to the exit, this is the message descriptor that would be returned to the application if no conversion were performed. It therefore contains the *MDFMT*, *MDENC*, and *MDCSI* of the unconverted message contained in *INBUF*.

Note: The *MQMD* parameter passed to the exit is always the most recent version of *MQMD* supported by the queue manager which invokes the exit. If the exit is intended to be portable between different environments, the exit should check the *MDVER* field in *MQMD* to verify that the fields that the exit needs to access are present in the structure.

On IBM i, the exit is passed a version-2 *MQMD*.

On output, the exit should change the *MDENC* and *MDCSI* fields to the values requested by the application, if conversion was successful; these changes will be reflected back to the application. Any other changes that the exit makes to the structure are ignored; they are not reflected back to the application.

If the exit returns *XROK* in the *DXRES* field of the *MQDXP* structure, but does not change the *MDENC* or *MDCSI* fields in the message descriptor, the queue manager returns for those fields the values that the corresponding fields in the *MQDXP* structure had on input to the exit.

INLEN (10-digit signed integer) - input

Length in bytes of *INBUF*.

This is the length of the input buffer *INBUF*, and specifies the number of bytes to be processed by the exit. *INLEN* is the lesser of the length of the message data before conversion, and the length of the buffer provided by the application on the *MQGET* call.

The value is always greater than zero.

INBUF (1-byte bit string x INLEN) - input

Buffer containing the unconverted message.

This contains the message data before conversion. If the exit is unable to convert the data, the queue manager returns the contents of this buffer to the application after the exit has completed.

Note: The exit should not alter *INBUF*; if this parameter is altered, the results are undefined.

OUTLEN (10-digit signed integer) - input

Length in bytes of *OUTBUF*.

This is the length of the output buffer *OUTBUF*, and is the same as the length of the buffer provided by the application on the *MQGET* call.

The value is always greater than zero.

OUTBUF (1-byte bit string x OUTLEN) - output

Buffer containing the converted message.

On output from the exit, if the conversion was successful (as indicated by the value *XROK* in the *DXRES* field of the *MQDXP* parameter), *OUTBUF* contains the message data to be delivered to the application, in the requested representation. If the conversion was unsuccessful, any changes that the exit has made to this buffer are ignored.

RPG invocation (ILE)

```
C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      exitname(MQDXP : MQMD : INLEN :
C                               INBUF : OUTLEN : OUTBUF)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
Dexitname      PR          EXTPROC('exitname')
D* Data-conversion exit parameter block
```

```

D MQDXP                                44A
D* Message descriptor
D MQMD                                364A
D* Length in bytes of INBUF
D INLEN                                10I 0 VALUE
D* Buffer containing the unconverted message
D INBUF                                *   VALUE
D* Length in bytes of OUTBUF
D OUTLEN                                10I 0 VALUE
D* Buffer containing the converted message
D OUTBUF                                *   VALUE

```

End of product-sensitive programming interface

SOAP reference

IBM MQ transport for SOAP reference information arranged alphabetically.

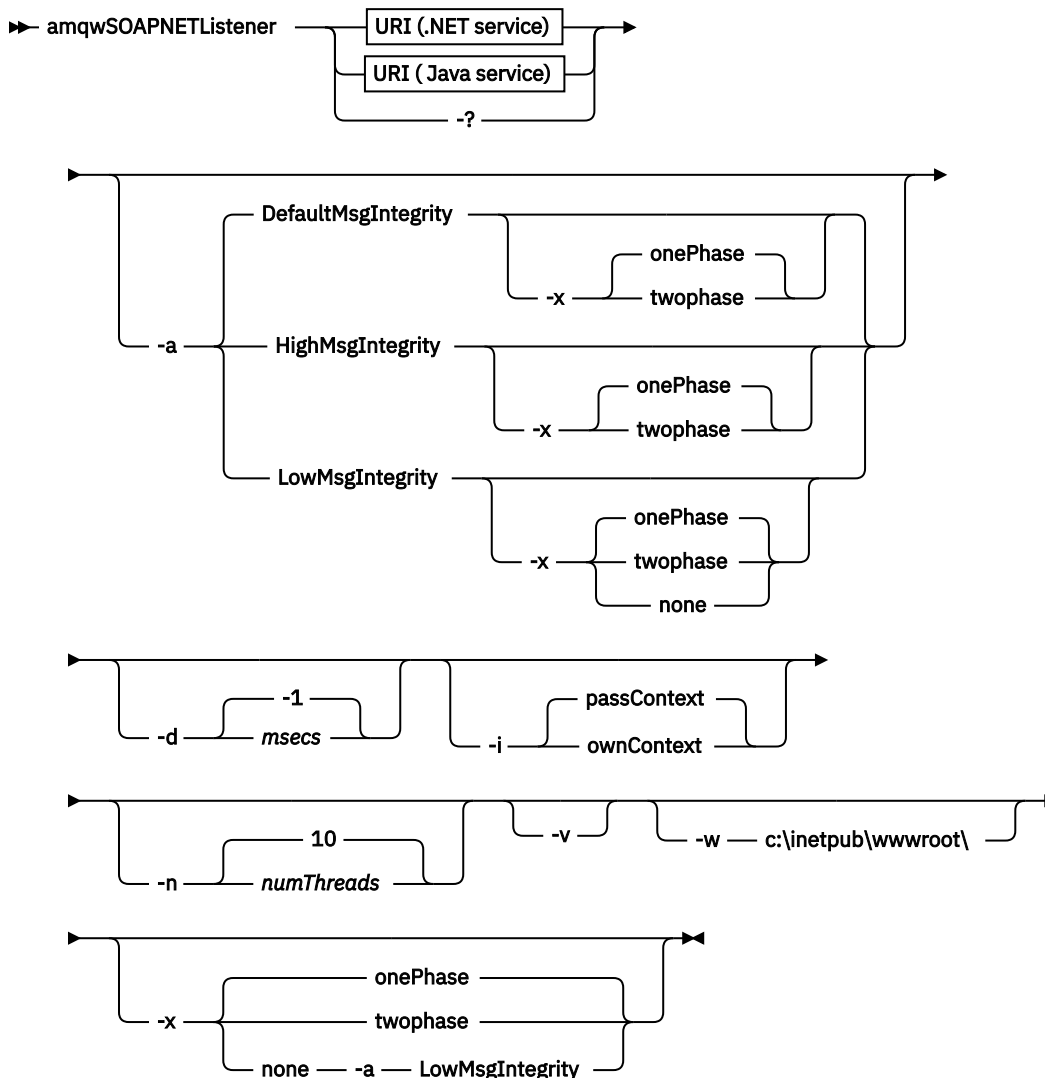
amqwSOAPNETListener: IBM MQ SOAP listener for .NET Framework 1 or 2

Syntax and parameters for the IBM MQ SOAP listener for .NET Framework 1 or 2.

Purpose

Starts the IBM MQ SOAP listener for .NET Framework 1 or 2.

.NET



Required parameters

URI *platform*

See [“URI syntax and parameters for Web service deployment”](#) on page 1447.

-?

Print out help text describing how the command is used.

Optional parameters

-a *integrityOption*

integrityOption specifies the behavior of IBM MQ SOAP listeners if it is not possible to put a failed request message on the dead-letter queue. *integrityOption* can take one of the following values:

DefaultMsgIntegrity

For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. *DefaultMsgIntegrity* applies if the -a option is omitted, or if *integrityOption* is not specified.

LowMsgIntegrity

For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity

For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits.

The deployment utility checks for the compatibility of the -x and -a flags. If -x none is specified, then -a LowMsgIntegrity must be specified. If the flags are incompatible the deployment utility exits with an error message and with no deployment steps having been undertaken.

-d *msecs*

msecs specifies the number of milliseconds for the IBM MQ SOAP listener to stay alive if request messages have been received on any thread. If *msecs* is set to -1, the listener stays alive indefinitely.

-i *Context*

Context specifies whether the listeners pass identity context. *Context* takes the following values:

passContext

Set the identity context of the original request message into the response message. The SOAP listener checks that it has authority to save the context from the request queue and to pass it to the response queue. It makes the checks at run time when opening the request queue to save context, and the response queue to pass context. If it does not have the required authority, or the MQOPEN call fails, and the response message is not processed. The response message is put on the dead-letter queue with the dead-letter header containing the return code from the failed MQOPEN. The listener then continues to process subsequent incoming messages as normal.

ownContext

The SOAP listener does not pass context. The returned context reflects the user ID under which the listener is running rather than the user ID which created the original request message.

The fields in the origin context are set by the queue manager, and not by the SOAP listener.

-n *numThreads*

numThreads specifies the number of threads in the generated startup scripts for the IBM MQ SOAP listener. The default is 10. Consider increasing this number if you have high message throughput.

-v

-v sets verbose output from external commands. Error messages are always displayed. Use -v to output commands that you can tailor to create customized deployment scripts.

-w *serviceDirectory*

serviceDirectory is the directory containing the web service.

-x *transactionality*

transactionality specifies the type of transactional control for the listener. *transactionality* can be set to one of the following values:

onePhase

IBM MQ one-phase support is used. If the system fails during processing, the request message is redelivered to the application. IBM MQ transactions ensure that the response messages are written exactly once.

twoPhase

Two-phase support is used. If the service is written appropriately the message is delivered exactly once, coordinated with other resources, within a single committed execution of the service. This option applies to server bindings connections only.

none

No transactional support. If the system fails during processing, the request message can be lost, even if persistent. The service might or might not have executed, and response, report or dead-letter messages might or might not be written.

The deployment utility checks for the compatibility of the -x and -a flags. See the description of the -a flag for details.

.NET Example

```
amqwSOAPNETlistener
-u "jms:/queue?destination=myQ&connectionFactory=()
&targetService=myService&initialContextFactory=com.ibm.mq.jms.Nojndi"
-w C:/wmqsoap/demos
-n 20
```

amqswsd1: generate WSDL for .NET Framework 1 or 2 service

amqswsd1 takes a Web service written for .NET Framework 1 or 2, and generates the WSDL for the class, inserting the URI you provide for the IBM MQ transport for SOAP into the generated WSDL.

Purpose

Use **amqswsd1** to generate WSDL containing the URI of the service deployed to IBM MQ. Use the WSDL to generate client proxies.

➡ **amqswsd1** — *escapedUri* — *className* — .asmx — *className* — .wsdl ➡

Parameters

***escapedUri* (Input)**

The URI of the service, with all "&" escaped to "&". For example:

```
"jms:/queue?destination=REQUESTDOTNET
&initialContextFactory=com.ibm.mq.jms.Nojndi
&connectionFactory=(connectQueueManager(QM1)binding(server))
&targetService=Quote.asmx"
```

***className.asmx* (Input)**

The service class.

***className.wsdl* (Output)**

The service WSDL.

Description

If the class is implemented using the code-behind programming model, you must build *className.dll* and store it in *./bin*.

amqwclientconfig: create Axis 1.4 Web services client deployment descriptor for IBM MQ transport for SOAP

amqwclientconfig creates the `client-config.wsdd` Axis 1.4 client deployment descriptor file.

Purpose

It adds the `jms:/transport` to the descriptor, and registers `java:com.ibm.mq.soap.transport.jms.WMQSender` as the class to handle SOAP requests for the `jms:transport`.

Syntax

➤ `amqwclientconfig` ➤

Description

amqwclientconfig calls **amqwsetcp** to set the CLASSPATH and runs the command:

```
java org.apache.axis.utils.Admin client "%WMQSOAP_HOME%\bin\amqwclientTransport.wsdd"
```

amqwdeployWMQService: deploy Web service utility

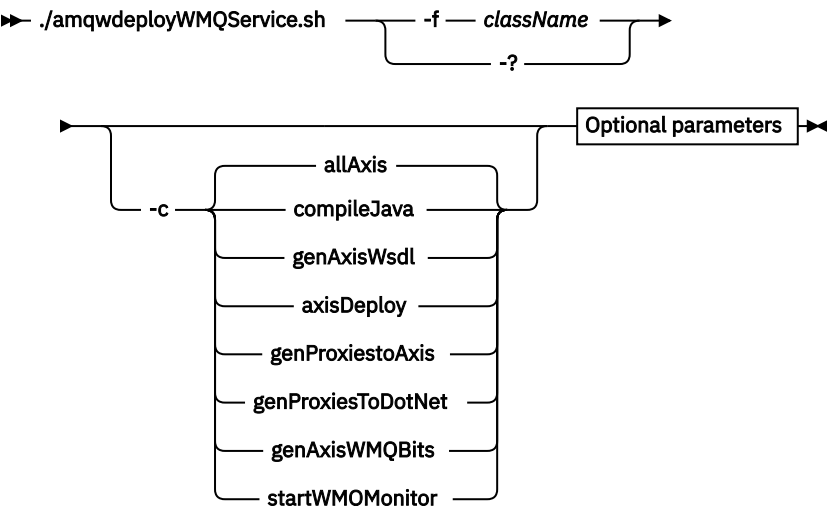
The deployment utility prepares a service class for use as a Web service using IBM MQ as the transport.

Purpose

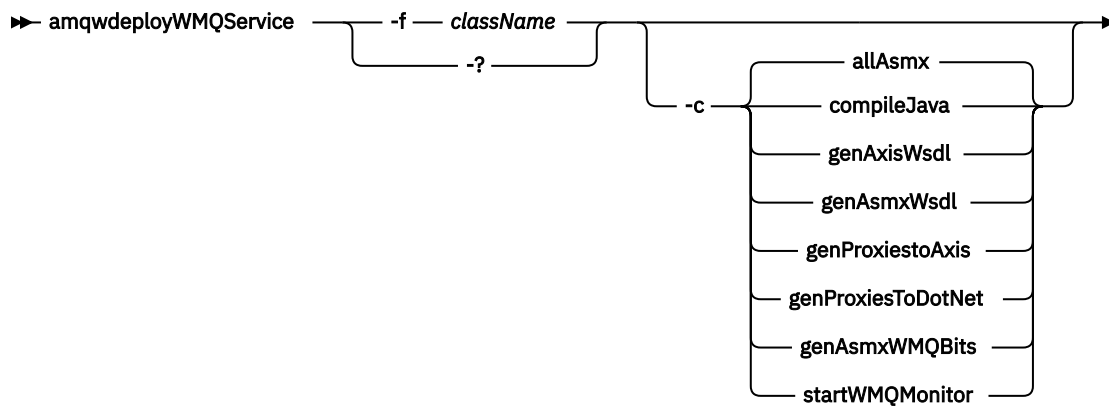
Use the deployment utility to generate the files that are needed to deploy an Axis 1.4, .NET Framework 1 or .NET Framework 2 service. Use the files to deploy a service invoked by IBM MQ. The files generated by **amqwdeployWMQService** are shown in [“Output files from amqwdeployWMQService” on page 1419](#).

Syntax diagram

UNIX and Linux systems

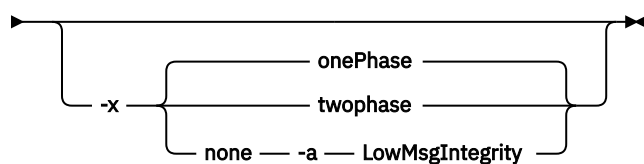
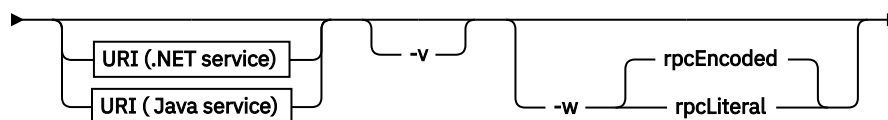
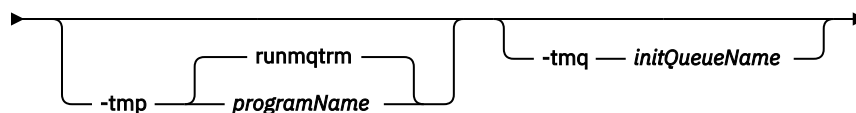
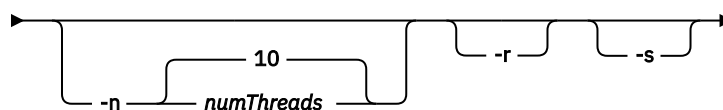
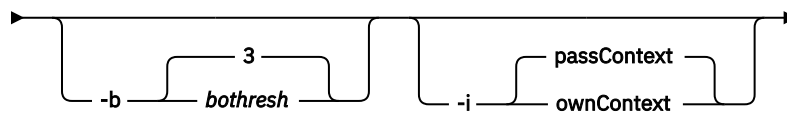
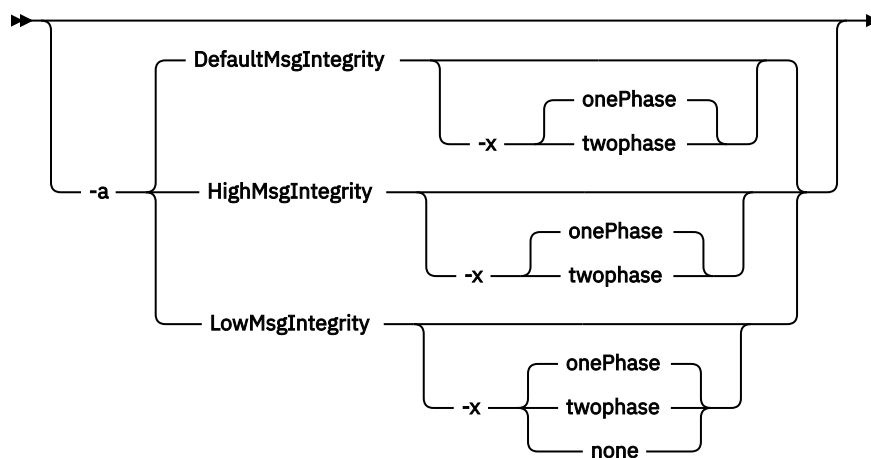


Windows



Optional parameters

Optional parameters



Required parameters

-f *className*

className is the name of the class to be deployed. For Axis services *className* is the Java source file, and for .NET services, the .asmx file. Figure 10 on page 1416 illustrates the deployment of an Axis service and Figure 11 on page 1416 of a .NET service.

```
amqwdeployMQService -f javaDemos/service/StockQuoteAxis.java
```

Figure 10. Example deployment of Axis service

```
amqwdeployMQService -f StockQuoteDotNet.asmx
```

Figure 11. Example deployment of .NET service

For Java, *className* must be fully qualified by the package name. It can be specified as a path name with directory separators or as a class name with period separators. The generated class is located in `./generated/client/remote/path name`. For a .NET service, although the directory can be specified, generated Java proxies are always located in `./generated/client/remote/dotNetService`.

If you specify a URI with the -u option and within the URI specify *targetService*, the deployment utility checks the *className*. *className* must match *targetService*. If the class and service do not match, the deployment utility displays an error message and exits.

-?

Print out help text describing how the command is used.

Optional parameters

-a *integrityOption*

integrityOption specifies the behavior of IBM MQ SOAP listeners if it is not possible to put a failed request message on the dead-letter queue. *integrityOption* can take one of the following values:

DefaultMsgIntegrity

For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. DefaultMsgIntegrity applies if the -a option is omitted, or if *integrityOption* is not specified.

LowMsgIntegrity

For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity

For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits.

The deployment utility checks for the compatibility of the -x and -a flags. If -x none is specified, then -a LowMsgIntegrity must be specified. If the flags are incompatible the deployment utility exits with an error message and with no deployment steps having been undertaken.

-b *bothresh*

bothresh specifies the backout threshold setting for the request queue. The default is 3.

-c *operation*

operation specifies which part of the deployment process to execute. *operation* is one of the following options:

allAxis

Perform all compile and setup steps for an Axis or Java service ⁵.

⁵ Default if *className* has a .java extension

compileJava

Compile the Java service: .java to .class.

genAxisWsd1

Generate WSDL: .class to .wsdl.

axisDeploy

Deploy the class file: .wsdl to .wsdd, apply .wsdd.

genProxiestoAxis

Generate proxies: .wsdl to .java and .class.

genAxisWMQBits

Set up IBM MQ queues, IBM MQ SOAP listeners and triggers for an Axis service.

allAsmx

Perform all setup steps for a .NET service ⁶.

genAsmxWsd1

Generate WSDL: .asmx to .wsdl.

genProxiesToDotNet

Generate proxies: .wsdl to .java, .class, .cs and .vb.

genAsmxWMQBits

Set up IBM MQ queues, IBM MQ SOAP listeners and triggers

startWMQMonitor

Start the trigger monitor for IBM MQ SOAP services.

Note: `runmqtrm` runs under the mqm user ID. If security is an issue, you must ensure that the listeners are started under appropriate user IDs.

-i Context

Context specifies whether the listeners pass identity context. *Context* takes the following values:

passContext

Set the identity context of the original request message into the response message. The SOAP listener checks that it has authority to save the context from the request queue and to pass it to the response queue. It makes the checks at run time when opening the request queue to save context, and the response queue to pass context. If it does not have the required authority, or the MQOPEN call fails, and the response message is not processed. The response message is put on the dead-letter queue with the dead-letter header containing the return code from the failed MQOPEN. The listener then continues to process subsequent incoming messages as normal.

ownContext

The SOAP listener does not pass context. The returned context reflects the user ID under which the listener is running rather than the user ID which created the original request message.

The fields in the origin context are set by the queue manager, and not by the SOAP listener.

-n numThreads

numThreads specifies the number of threads in the generated startup scripts for the IBM MQ SOAP listener. The default is 10. Consider increasing this number if you have high message throughput.

-r

-r specifies that any existing request or trigger monitor queue definitions are replaced. Trigger monitor queues are replaced only if *-tmq* is also specified. Queues are re-created with standard default attributes and existing messages on the queues are deleted. If the *-r* option is not used then any existing queue definitions are not altered and existing messages are not deleted. By not specifying *-r*, you ensure that any customized queue attributes are preserved.

-s

Configure the listener to run as an IBM MQ service. If *-s* and *-tmq* are both specified, the deployment utility displays an error message and exits.

⁶ Default if *className* has a .asmx extension.

-tmp *programName*

programName specifies the name of a trigger monitor program. Use `-tmp programName` in a UNIX or Linux environment as an alternative to using **runmqtrm**. Programs it initiates run under mqm authority.

For example:

```
amqwdployWmqService -f javaDemos/service/StockQuoteAxis.java
-tmq trigger.monitor.queue -tmp trigmon
```

-tmq *queueName*

queueName specifies a trigger monitor queue name. IBM MQ process definitions are created to configure automatic triggering of IBM MQ SOAP listeners with the associated trigger monitor queue name. If the option is not specified then no triggering configuration is defined by the deployment utility. If `-s` and `-tmq` are both specified, the deployment utility displays an error message and exits.

URI *platform*

See [“URI syntax and parameters for Web service deployment” on page 1447](#).

-v

`-v` sets verbose output from external commands. Error messages are always displayed. Use `-v` to output commands that you can tailor to create customized deployment scripts.

-w

`-w` controls the style of WSDL to generate. The default is `rpcEnclosed`, for compatibility with previous releases of IBM MQ transport for SOAP. Use `rpcLiteral` to create WSDL compatible with Axis2 client proxy generation. `rpcEnclosed` is not compatible with WS-I recommendations.

-x *transactionality*

transactionality specifies the type of transactional control for the listener. *transactionality* can be set to one of the following values:

onePhase

IBM MQ one-phase support is used. If the system fails during processing, the request message is redelivered to the application. IBM MQ transactions ensure that the response messages are written exactly once.

twoPhase

Two-phase support is used. If the service is written appropriately the message is delivered exactly once, coordinated with other resources, within a single committed execution of the service. This option applies to server bindings connections only.

none

No transactional support. If the system fails during processing, the request message can be lost, even if persistent. The service might or might not have executed, and response, report or dead-letter messages might or might not be written.

The deployment utility checks for the compatibility of the `-x` and `-a` flags. See the description of the `-a` flag for details.

Errors

On Windows, if errors are reported from **amqswsd1**, try issuing the following command to register `.asmx` files as services.

```
%windir%\Microsoft.NET\Framework/ version number /aspnet_regiis.exe -ir
```

The problem typically occurs on systems where IIS has not been installed, or IIS has been installed after .NET. The problem is encountered when **amqswsd1** generates the `.wsdl` files.

Note: The registry keys are also required to permit the listener to invoke the services. If you use your own customized deployment procedures, you might not encounter the problem until run time.

Output files from amqwdeployWMQService

A list of the directories and files output from **amqwdeployWMQService**

Table 633. Output files from amqwdeployWMQService			
Output s	Descriptio n	Output directory	Filename
.class	Compiled Java source file	./generated/server/server package	classname.class
.wsdl	Service description	./generated	className Axis_Wmq.wsdl className DotNet_Wmq.wsdl
.wsdd	Axis client and service deployment files	./	client-config.wsdd server-config.wsdd
		./generated/server/server package	className _deploy.wsdd className _undeploy.wsdd
Client source (.vb, .cs, .java)	.Net client stubs to Axis service	./generated/client	className AxisService.cs className AxisService.vb
	.Net client stubs to .Net service	./generated/client	className DotNet.cs className DotNet.vb
Client helper (.java and .class)	Java client proxies to .Net service	./generated/server/soap/ client/ remote/dotnetService	className DotNet.class className DotNet.java className DotNetLocator.class className DotNetLocator.java className DotNetSoap12Stub.class className DotNetSoap12Stub.java className DotNetSoap_BindingStub.class className DotNetSoap_BindingStub.java className DotNetSoap_PortType.class className DotNetSoap_PortType.java
	Java client proxies to Axis service	./generated/server/soap/ client/ remote/client package	SoapServer className AxisBindingSoapStub.class SoapServer className AxisBindingSoapStub.java className Axis.class className Axis.java className AxisService.class className AxisService.java className AxisServiceLocator.class className AxisServiceLocator.java

Table 633. Output files from amqwdeployWMQService (continued)			
Output s	Descriptio n	Output directory	Filename
Scripts (.cmd and .sh)	Listener scripts	/generated/server	startWMQJListener.cmd startWMQJListener.sh startWMQNListener.cmd endWMQJListener.cmd endWMQJListener.sh endWMQNListener.cmd

Usage notes for amqwdeployWMQService

Describes the tasks performed by **amqwdeployWMQService**.

The deployment utility performs the following actions.

1. Checks paths to the following files:

- axis.jar.
- WMQSOAP_HOME/java/lib/com.ibm.mq.soap.jar.
- On Windows, csc.exe

2. On Windows, uses either %SystemRoot%\Microsoft.NET\Framework\v1.1.432 or, if the C# compiler is installed, the path to csc.exe as the path to the .NET Framework.

Note: If you have Microsoft Visual Studio 2008 installed (Version 9), wsdl.exe is not in the path to csc.exe. You need to add the path to the .NET framework to your Path variable; for example:

```
Set Path=C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727;%Path%
```

3. Creates the ./generated directory, and required subdirectories, if they do not exist.

4. For Java services, compiles the source into *className.class*.

5. Generates WSDL.

6. For Java services, creates deployment descriptor files *className_deploy.wsdd* and *className_undeploy.wsdd*

7. For Java services, creates or updates the Axis deployment descriptor file, server-config.wsdd.

8. Generates the client proxies for Java, C# and Visual Basic from the WSDL.

Note: On Windows, the deploy utility generates proxies for Visual Basic and C# regardless of the language in which the service is written. The WSDL and the proxies generated from it include the appropriate URI to call the service:

```
a. jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM
&connectionFactory=(connectQueueManager(WMQSOAP.DEMO.QM))
&initialContextFactory=com.ibm.mq.jms.Nojndi
&targetService=StockQuoteDotNet.asmx
&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
```

Figure 12. Example URI in generated .NET client to call .NET service

```
b. jms:/queue?destination=SOAPJ.demos@WMQSOAP.DEMO.QM
&connectionFactory=(connectQueueManager(WMQSOAP.DEMO.QM))
&initialContextFactory=com.ibm.mq.jms.Nojndi
&targetService=soap.server.StockQuoteAxis.java
&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
```

Figure 13. Example URI in generated .NET client to call Axis 1 service

9. Compiles the Java proxies.
 10. Creates an IBM MQ queue, *requestQueue* to hold requests for the service. The default queue name is of the form SOAPJ. *directory* , or you can specify *requestQueue* in the -u URI option.
 11. Creates command and shell script files to start the IBM MQ SOAP listeners that process the request queue.
 12. If the -tmq option has been used, the deployment utility creates IBM MQ definitions to trigger IBM MQ SOAP listener processes automatically.
 - The deployment utility uses the APPLICID attribute of the **runmqsc** DEFINE PROCESS command to contain a command to start the listener. The command has the name of the deployment directory embedded in it. The APPLICID field has a maximum length of 256, which limits the maximum length of the deployment directory. The directory limit for Java services is as follows:
 - UNIX and Linux systems: 218
 - Windows: 197 minus the length of the request queue name.
- For .NET services, the directory limit is as follows:
- Windows: 209 minus length of the service name, minus the .asmx extension.
- The deployment utility checks whether the limit for APPLICID is exceeded. If the limit is exceeded, the utility does not attempt to define the triggering process. It displays an error message and the deployment process fails without performing any deployment steps.

The following examples show the configuration and start commands generated by the deployment utility to start an IBM MQ SOAP listener.

```
DEFINE PROCESS(requestQueue) APPLICID(applicIDStr) REPLACE
ALTER QLOCAL (requestQueue) TRIGTYPE(FIRST) TRIGGER
PROCESS(requestQueue) INITQ(initQueueName) TRIGMPRI(0)
```

Figure 14. IBM MQ configuration commands to trigger a SOAP listener.

```
applicIDStr = start "Java WMQSoapListener -requestQueue"
               /min .\generated\server\startWMQJListener.cmd;
```

Figure 15. Starting Axis SOAP listener on Windows

```
applicIDStr = start "WMQAsmxListener -className\
               /min .\generated\server\startWMQNListener.cmd;
```

Figure 16. Starting .NET SOAP listener on Windows

```
applicIDStr = xterm -iconic -T \"Java WMQSoapListener_requestQueue\"
               -e ./generated/server/startWMQJListener.sh & #
```

Figure 17. Starting Axis SOAP listener on UNIX and Linux systems

amqwRegisterdotNet: register IBM MQ transport for SOAP to .NET

Register IBM MQ transport for SOAP to the global assembly cache on .NET.

Purpose

amqwRegisterdotNet registers the IBM MQ SOAP sender, SOAP listener, and WSDL processor with .NET Framework 1 or 2.

Syntax

➤ amqwRegisterdotNet ➤

Description

amqwRegisterdotNet is run automatically during installation. You do not need to run it again if the .NET Framework you are using was installed before IBM MQ transport for SOAP. You can run it as many times as you want. Use it to reregister IBM MQ transport for SOAP with different .NET Framework versions.

Note: On Windows 2003 Server, you must also run the **aspnet_regiis** utility, even if you are not deploying to Internet Information Server (IIS). The location of the **aspnet_regiis.exe** utility might vary with different versions of the Microsoft.NET Framework, but it is typically located in: %SystemRoot%/Microsoft.NET/Framework/version number/aspnet_regiis. If multiple versions are installed, use **aspnet_regiis** for the version of .NET Framework you are using.

Apache software license

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>
<https://www.apache.org/licenses/>

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the

editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special,

incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

MQMD SOAP settings

The IBM MQ SOAP sender and IBM MQ SOAP listener create a message descriptor (**MQMD**). The topic describes the fields you must set in the MQMD if you create your own SOAP sender or listener.

Purpose

The values set in the **MQMD** control the exchange of messages between the IBM MQ SOAP sender, the IBM MQ SOAP listener, and the SOAP client program. If you create your own SOAP sender or listener, follow the rules in [Table 634 on page 1426](#).

Description

Table 634 on page 1426 describes how the **MQMD** fields are set by the IBM MQ SOAP sender and IBM MQ SOAP listener. If you write your own sender or listener you must set these fields in accordance with the rules for exchanging messages. The IBM MQ SOAP listener conforms to typical IBM MQ message exchange protocols. If you write your own sender to work with the IBM MQ SOAP listeners, you can set different **MQMD** values.

In Table 634 on page 1426, the values in the Setting column are organized as follows:

Request, One way

Settings made by IBM MQ SOAP sender.

Response, Report

Settings made by IBM MQ SOAP listener in response to IBM MQ SOAP sender request.

ALL

Settings made by both the IBM MQ SOAP sender and IBM MQ SOAP listener.

Customized sender

You can write your own sender. Typically, a customized sender overrides the standard report options.

Table 634. MQMD SOAP settings		
Field name	Setting	Values
<i>StrucId</i>	ALL MQMD_STRUC_ID	'MD-1' 1
<i>Version</i>	ALL MQMD_VERSION_2	2
<i>Report</i>	ALL MQRO_NONE + MQRO_NEW_MSG_ID + MQRO_COPY_MSG_ID_TO_CORREL_ID + MQRO_EXCEPTION + MQRO_EXPIRY + MQRO_DISCARD Customized sender See “Customized report options” on page 1430	52428800
<i>MsgType</i>	Request MQMT_REQUEST Response MQMT_REPLY Report MQMT_REPORT One way MQMT_DATAGRAM	MQMT_REQUEST 1 MQMT_REPLY 2 MQMT_REPORT 4 MQMT_DATAGRAM 8
<i>Expiry</i>	Request, One way Specified by Expiry option in URI. Defaults to MQEI_UNLIMITED. Response Value of Expiry in request message Report MQEI_UNLIMITED	MQEI_UNLIMITED -1

Table 634. MQMD SOAP settings (continued)

Field name	Setting	Values
<i>Feedback</i>	<p>Request, Response, One way MQFB_NONE.</p> <p>Report</p> <ul style="list-style-type: none"> Generated by queue manager - value set according to normal rules. Generated by IBM MQ SOAP Listener: <p>MQRC_BACKOUT_THRESHOLD_REACHED Backout threshold for multiple attempts exceeded.</p> <p>MQRCCF_MD_FORMAT_ERROR The message is not recognized as having an MQRFH2 header.</p> <p>MQRC_RFH_PARM_MISSING A required parameter, for example, SoapAction, in MQRFH2 is missing.</p> <p>MQRC_RFH_FORMAT_ERROR A basic integrity check of the MQRFH2 failed, for example, internal lengths are corrupted.</p> <p>MQRC_RFH_ERROR The MQRFH2 passed an integrity check, but the body of the message is not set to MQFMT_NONE.</p>	<p>MQFB_NONE 0</p> <p>MQRC_BACKOUT_THRESHOLD_REACHED 2362</p> <p>MQRCCF_MD_FORMAT_ERROR 3023</p> <p>MQRC_RFH_PARM_MISSING 2339</p> <p>MQRC_RFH_FORMAT_ERROR 2421</p> <p>MQRC_RFH_ERROR 2334</p>
<i>Encoding</i>	<p>ALL MQENC_NATIVE</p>	Depends on environment
<i>CodedCharSetId</i>	<p>ALL Set to UTF-8</p>	1208
<i>Format</i>	<p>Request, Response, One way MQFMT_RF_HEADER_2</p> <p>Report</p> <p>Queue manager reports Follows IBM MQ rules</p> <p>IBM MQ SOAP listener reports Format of the original request message.</p>	<p>MQFMT_RF_HEADER_2 "MQRFH2 "</p>
<i>Priority</i>	<p>Request, One way Specified by Priority option in URI. Defaults to MQPRI_PRIORITY_AS_Q_DEF.</p> <p>Response, Report Value of Priority in the request message.</p>	<p>MQPRI_PRIORITY_AS_Q_DEF -1</p>

Table 634. MQMD SOAP settings (continued)

Field name	Setting	Values
<i>Persistence</i>	Request, One way MQPER_PERSISTENCE_AS_Q_DEF. Response, Report Value of Persistence in the request message.	MQPER_PERSISTENCE_AS_Q_DEF 2
<i>MsgId</i>	Request, One way Generated by the queue manager. Response, Report The IBM MQ SOAP sender sets MQRO_NEW_MSG_ID and <i>MsgId</i> is generated	Generated Unique value generated by the queue manager
<i>CorrelId</i>	Request, One way, Report MQCI_NONE Response, Report The IBM MQ SOAP sender sets MQRO_COPY_MSG_ID_TO_CORREL_ID and the listener copies <i>MsgId</i> from the request message.	MQCI_NONE 0
<i>BackoutCount</i>	ALL Not used	0
<i>ReplyToQ</i>	Request Specified by replyDestination option in URI. Defaults to SYSTEM.SOAP.RESPONSE.QUEUE. Response, One way, Report Left blank	
<i>ReplyToQMGr</i>	ALL Field left blank	Generated by the queue manager; see Reply-to queue and queue manager .
<i>UserIdentifier</i>	Request, One way, Report Left blank Response Depends on the -i <i>passContext</i> option supplied to the listener, and the authority the listener is running under.	Request, One way, Report Generated by the queue manager; see “ UserIdentifier (MQCHAR12) ” on page 441 Response Variable
<i>AccountingToken</i>	ALL MQACT_NONE	MQACT_NONE Null string or blanks Set by the queue manager; see “ AccountingToken (MQBYTE32) ” on page 400
<i>ApplIdentityData</i>	ALL None	Null string or blanks ²

Table 634. MQMD SOAP settings (continued)

Field name	Setting	Values
<i>PutApplType</i>	ALL MQAT_NO_CONTEXT	MQAT_NO_CONTEXT 0 Value generated by queue manager; see “PutApplType (MQLONG)” on page 428.
<i>PutApplName</i>	ALL None	Value generated by queue manager; see “PutApplName (MQCHAR28)” on page 427.
<i>PutDate</i>	ALL None	Value generated by queue manager; see “PutDate (MQCHAR8)” on page 429.
<i>PutTime</i>	ALL None	Value generated by queue manager; see “PutTime (MQCHAR8)” on page 430.
<i>ApplOriginData</i>	ALL None	Null string or blanks ²
<i>GroupId</i>	Request, One way, Report MQGI_NONE Response Field is copied from the request message	Nulls
<i>MsgSeqNumber</i>	Request, One way, Report Not used Response Field is copied from the request message	Generated by the queue manager; see Physical order on a queue .
<i>Offset</i>	Request, One way, Report Not used Response Field is copied from the request message	0
<i>MsgFlags</i>	Request, One way, Report MQMF_NONE Response Field is copied from the request message	MQMF_NONE 0 See “MsgFlags (MQLONG)” on page 417
<i>OriginalLength</i>	Request, One way, Response MQOL_UNDEFINED Report Length of original request message	MQOL_UNDEFINED -1

Table 634. MQMD SOAP settings (continued)

Field name	Setting	Values
Notes: <ol style="list-style-type: none"> 1. The symbol ~ represents a single blank character. 2. The value Null string or blanks denotes the null string in C, and blank characters in other programming languages. 		

Customized report options

You can write your own SOAP sender and use it with the supplied listeners. Typically you might write a sender to change the choice of report options. The IBM MQ SOAP listeners support most combinations of report options, as described in the following lists.

- Report options supported by IBM MQ SOAP listeners:
 - MQRO_EXCEPTION
 - MQRO_EXCEPTION_WITH_DATA
 - MQRO_EXCEPTION_WITH_FULL_DATA
 - MQRO_DEAD_LETTER_Q
 - MQRO_DISCARD_MSG
 - MQRO_NONE
 - MQRO_NEW_MSG_ID
 - MQRO_PASS_MSG_ID
 - MQRO_COPY_MSG_ID_TO_CORREL_ID
 - MQRO_PASS_CORREL_ID
- Report options supported by the queue manager:
 - MQRO_COA
 - MQRO_COA_WITH_DATA
 - MQRO_COA_WITH_FULL_DATA
 - MQRO_COD
 - MQRO_COD_WITH_DATA
 - MQRO_COD_WITH_FULL_DATA
 - MQRO_EXPIRATION
 - MQRO_EXPIRATION_WITH_DATA
 - MQRO_EXPIRATION_WITH_FULL_DATA
- The following report options are not supported by the IBM MQ SOAP listeners.
 - MQRO_PAN
 - MQRO_NAN

The behavior of IBM MQ SOAP listeners in response to combinations of MQRO_EXCEPTION_* and MQRO_DISCARD is described in [Table 635 on page 1431](#).

The notation MQRO_EXCEPTION_* indicates the use of either MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA or MQRO_EXCEPTION_WITH_FULL_DATA.

runivt: IBM MQ transport for SOAP installation verification test

An installation verification test suite (IVT) is provided with IBM MQ transport for SOAP. **runivt** runs a number of demonstration applications and ensures that the environment is correctly set up after installation.

Purpose

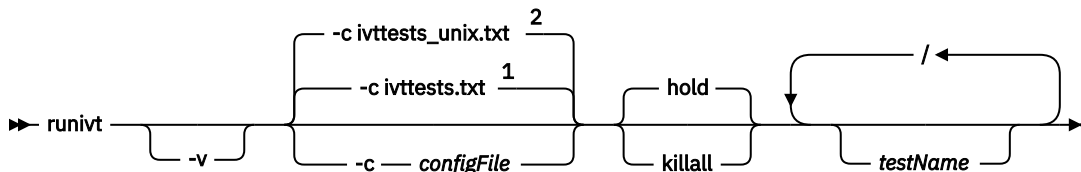
The **runivt** command uses the sample programs provided with the IBM MQ transport for SOAP to send Web service requests from clients to services. It runs tests for Axis 1.4, .NET Framework 1, and .NET Framework 2. The tests are configured in a test script file. The default test script file for Windows runs a combination of tests between Java and .NET clients and services.

Description

runivt must be run from its own directory.

The command starts listeners in a different command window. For this reason, you must run the command from an X Window System session on UNIX and Linux systems.

runivt syntax



Notes:

¹ Default on Windows

² Default on UNIX and Linux systems

runivt parameters

-v

Verbose mode. Write fuller error messages to the console.

-c configFile

A configuration file defining the tests to be run. The default configuration file supplied with Windows, UNIX or Linux systems is used by default.

hold

Leave the listener running after the tests complete

killall

End the listener when the tests complete

testName

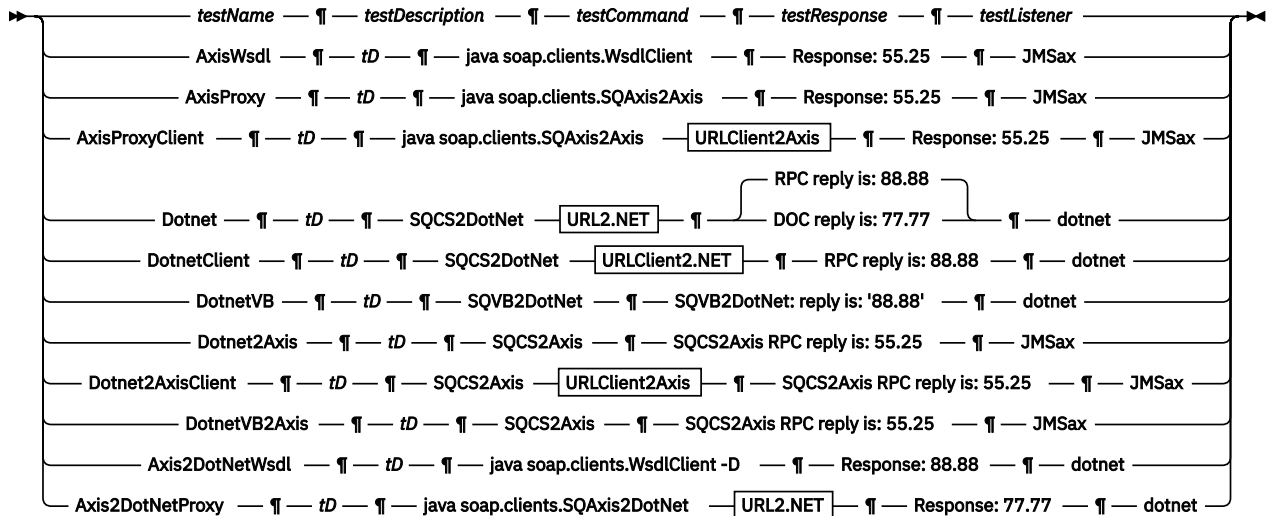
A space separated list of the tests to run. The test names are selected from the configuration file. If no names are specified then all the tests in the configuration file are run.

Configuration file

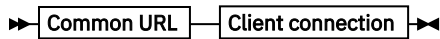
Each configuration file parameter is a separate line of the file. Leave a blank line between each group of parameters.

The parameters in the `ivttests.txt` parameter file are listed.

configFile syntax



URLClient2Axis



URL2.NET



URLClient2.NET



Common URL

```

jms:/queue?destination=SOAPJ.demos@WMQSOAP.DEMO.QM — & — initialContextFactory — = —>
com.ibm.mq.jms.Nojndi — & — connectionFactory — = —>
connectQueueManager — ( — WMQSOAP.DEMO.QM — ) —>
  
```

Client connection

```

clientConnection — ( — localhost%25289414WMQSOAP.DEMO.QM%2529 — ) — clientChannel —>
( — TESTCHANNEL — ) —>
  
```

Target service

```

& — targetService — = — StockQuoteDotNet.asmx —>
  
```

configFile parameters

testName

The name of the test. Use *testName* in the **runivt** command

testDescription

Documentation about the test

testCommand

The command executed by the **runivt** command to make the client request.

testResponse

The exact response string returned by the client request to the console. For the test to succeed *testResponse* must match the actual response.

testListener

The name of the IBM MQ SOAP listener that is started by **runivt** to process the SOAP request. **dotnet** and **JMSax** are synonyms for the supplied listeners, **amqwSOAPNETlistener** and **SimpleJavaListener**.

Examples

```
runivt
```

Figure 18. run all the default tests

```
runivt dotnet
```

Figure 19. run a specific test from the default tests

```
runivt -c mytests.txt
```

Figure 20. run a set of custom tests

Related tasks

[Verifying the IBM MQ transport for SOAP](#)

Secure Web services over IBM MQ transport for SOAP

You might secure Web services that use the IBM MQ transport for SOAP in one of two ways. Either create an SSL channel between the client and server, or use Web services security.

SSL and the IBM MQ transport for SOAP

The IBM MQ transport for SOAP provides several SSL options that can be specified for use with client channel configured to run in SSL mode. The options differ between the .NET and Java environments. The IBM MQ SOAP senders and listeners process only the SSL options that are applicable to their particular environment. They ignore options that are not applicable.

The presence or absence of the `sslCipherSpec` option for .NET clients and the `sslCipherSuite` option for Java clients determines whether SSL is used or not. If the option is not specified in the URI then by default SSL is not used and all other SSL options are ignored. All SSL options are optional except where indicated.

For IBM MQ clients, set the SSL attributes in the URI or channel definition table. On the server, set the attributes using the facilities of IBM MQ.

By default, the standard IBM MQ SSL option, `SSLCAUTH`, is set when enabling SSL on the channel. Clients must authenticate themselves before SSL communication can commence. If `SSLCAUTH` is not set, SSL communications are established without client authentication.

To authenticate themselves, clients must have a certificate assigned in their key repository that is acceptable to the queue manager. For additional security, IBM MQ channels can be configured to accept only certificates from a restricted list. The list is restricted by checking the distinguished name of the certificate against the peer name attribute of the channel.

If you use Java, the first SSL connection from an IBM MQ SOAP client causes the following SSL parameters to be fixed. The same values are used in subsequent connections using the same client process:

- `sslKeyStore`
- `sslKeyStorePassword`
- `sslTrustStore`
- `sslTrustStorePassword`

- `sslFipsRequired`
- `sslLDAPCRLservers`

The effect of varying these parameters on subsequent connections from this client is undefined.

If you use .NET, the first SSL connection from an IBM MQ SOAP client causes the following SSL parameters to be fixed. The same values are used in subsequent connections using the same client process:

- `sslKeyRepository`
- `sslCryptoHardware`
- `sslFipsRequired`
- `sslLDAPCRLservers`

The effect of varying these parameters on subsequent connections from this client is undefined. These parameters are reset if all SSL connections become inactive and a new SSL connection is made.

The following properties can also be specified as system properties:

- `sslKeyStore`
- `sslKeyStorePassword`
- `sslTrustStore`
- `sslTrustStorePassword`

If they are specified both as system properties and in the URI, and the values differ, the deployment utility displays a warning. The URI values take precedence.

Related tasks

[Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client](#)

Related reference

[SSL connection factory parameters in the IBM MQ Web services URI](#)

Add SSL options to the list of connection factory options in the IBM MQ Web services URI.

[Federal Information Processing Standards \(FIPS\) for UNIX, Linux and Windows](#)

SSL connection factory parameters in the IBM MQ Web services URI

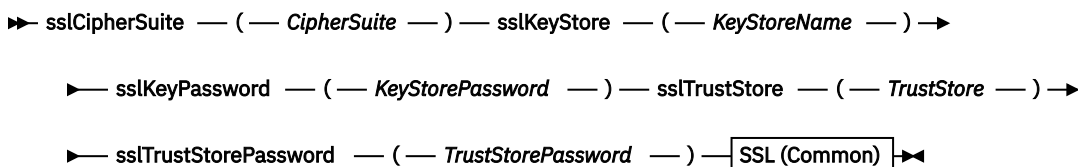
Add SSL options to the list of connection factory options in the IBM MQ Web services URI.

Purpose

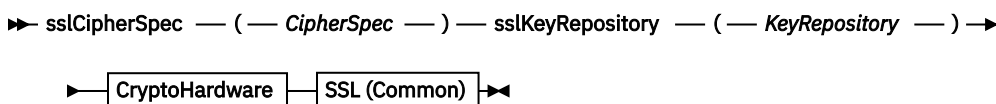
You can use a secure connection between an IBM MQ Web services client and the queue manager hosting the web service. The SSL options control how SSL is configured on the IBM MQ MQI client-server channel connection.

Syntax diagram

SSL (Java)

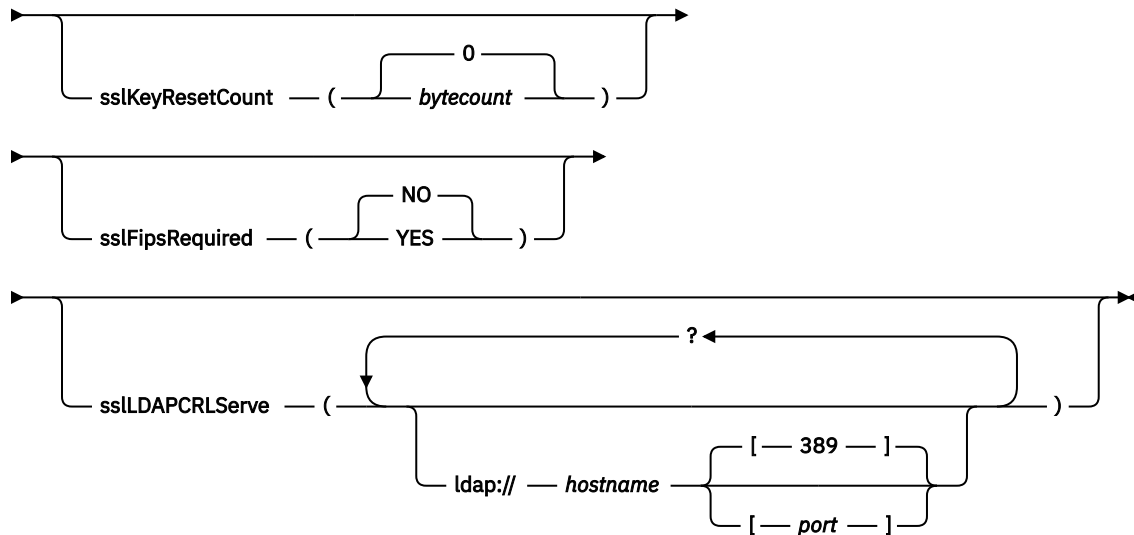


SSL (.NET)



SSL (Common)

➤ `sslCipherPeerName` — (— *PeerName* —) ➔



CryptoHardware

➤ `sslCryptoHardware` — = — *PKCS #11 Path and file name* — ; — *PKCS #11 token label* — ; ➔

➤ *PKCS #11 token password* — ; — *symmetric cipher setting* — ; ➤

Required SSL parameters (Common)

`sslPeerName` (*peerName*)

peerName specifies the `sslPeerName` used on the channel.

Required SSL parameters (Java)

`sslCipherSuite` (*CipherSuite*)

CipherSuite specifies the `sslCipherSuite` used on the channel. The `CipherSuite` specified by the client must match the `CipherSuite` specified on the server connection channel.

`sslKeyStore` (*KeyStoreName*)

KeyStoreName specifies the `sslKeyStoreName` used on the channel. The keystore holds the private key of the client used to authenticate the client to the server. The keystore is optional if the SSL connection is configured to accept anonymous client connections.

`sslKeyStorePassword` (*KeyStorePassword*)

KeyStorePassword specifies the `sslKeyStorePassword` used on the channel.

`sslTrustStore` (*TrustStoreName*)

TrustStoreName specifies the `sslTrustStoreName` used on the channel. The truststore holds the public certificate of the server, or its key chain, to authenticate the server to the client. The truststore is optional if the root certificate of a certificate authority is used to authenticate the server. In Java, root certificates are held in the JRE certificate store, `cacerts`.

`sslTrustStorePassword` (*TrustStorePassword*)

TrustStorePassword specifies the `sslTrustStorePassword` used on the channel.

Required SSL parameters (.NET)

`sslCipherSpec` (*CipherSpec*)

CipherSpec specifies the `sslCipherSpec` used on the channel. If the option is specified then SSL is used on the client channel.

sslKeyRepository (*KeyRepository*)

KeyRepository specifies the sslCipherSpec used on the channel where SSL keys and certificates are stored. *KeyRepository* is specified in stem format, that is, a full path with file name but with the file extension omitted. The effect of setting sslKeyRepository is the same as setting the KeyRepository field in the **MQSCO** structure on an MQCONN call.

Optional SSL parameters (.NET)**sslCryptoHardware (*CryptoHardware*)**

CryptoHardware specifies the sslCryptoHardware used on the channel. The possible values for this field, and the effect of setting it, are the same as for the CryptoHardware field of the **MQSCO** structure on an MQCONN.

Optional SSL parameters (Common)**sslKeyResetCount (*bytecount*)**

bytecount specifies the number of bytes passed across an SSL channel before the SSL secret key must be renegotiated. To disable the renegotiation of SSL keys omit the field or set it to zero. Zero is the only value supported in some environments, see [Renegotiating the secret key in IBM MQ classes for Java](#) . The effect of setting sslKeyResetCount is the same as setting the KeyResetCount field in the **MQSCO** structure on an MQCONN call.

sslFipsRequired (*fipsCertified*)

fipsCertified specifies whether *CipherSpec* or *CipherSuite* must use FIPS-certified cryptography in IBM MQ on the channel. The effect of setting *fipsCertified* is the same as setting the FipsRequired field of the **MQSCO** structure on an MQCONN call.

sslLDAPCRLServers (*LDAPServerList*)

LDAPServerList specifies a list of LDAP servers to be used for Certificate Revocation List checking.

For SSL enabled client connections, *LDAPServerList* is a list of LDAP servers to be used for Certificate Revocation List (CRL) checking. The certificate provided by the queue manager is checked against one of the listed LDAP CRL servers; if found, the connection fails. Each LDAP server is tried in turn until connectivity is established to one of them. If it is impossible to connect to any of the servers, the certificate is rejected. Once a connection has been successfully established to one of them, the certificate is accepted or rejected depending on the CRLs present on that LDAP server.

If *LDAPServerList* is blank, the certificate belonging to the queue manager is not checked against a Certificate Revocation List. An error message is displayed if the supplied list of LDAP URIs is not valid. The effect of setting this field is the same as that of including MQAIR records and accessing them from an **MQSCO** structure on an MQCONN.

Related tasks

[Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client](#)

Related reference

[SSL and the IBM MQ transport for SOAP](#)

The IBM MQ transport for SOAP provides several SSL options that can be specified for use with client channel configured to run in SSL mode. The options differ between the .NET and Java environments. The IBM MQ SOAP senders and listeners process only the SSL options that are applicable to their particular environment. They ignore options that are not applicable.

[Federal Information Processing Standards \(FIPS\) for UNIX, Linux and Windows](#)

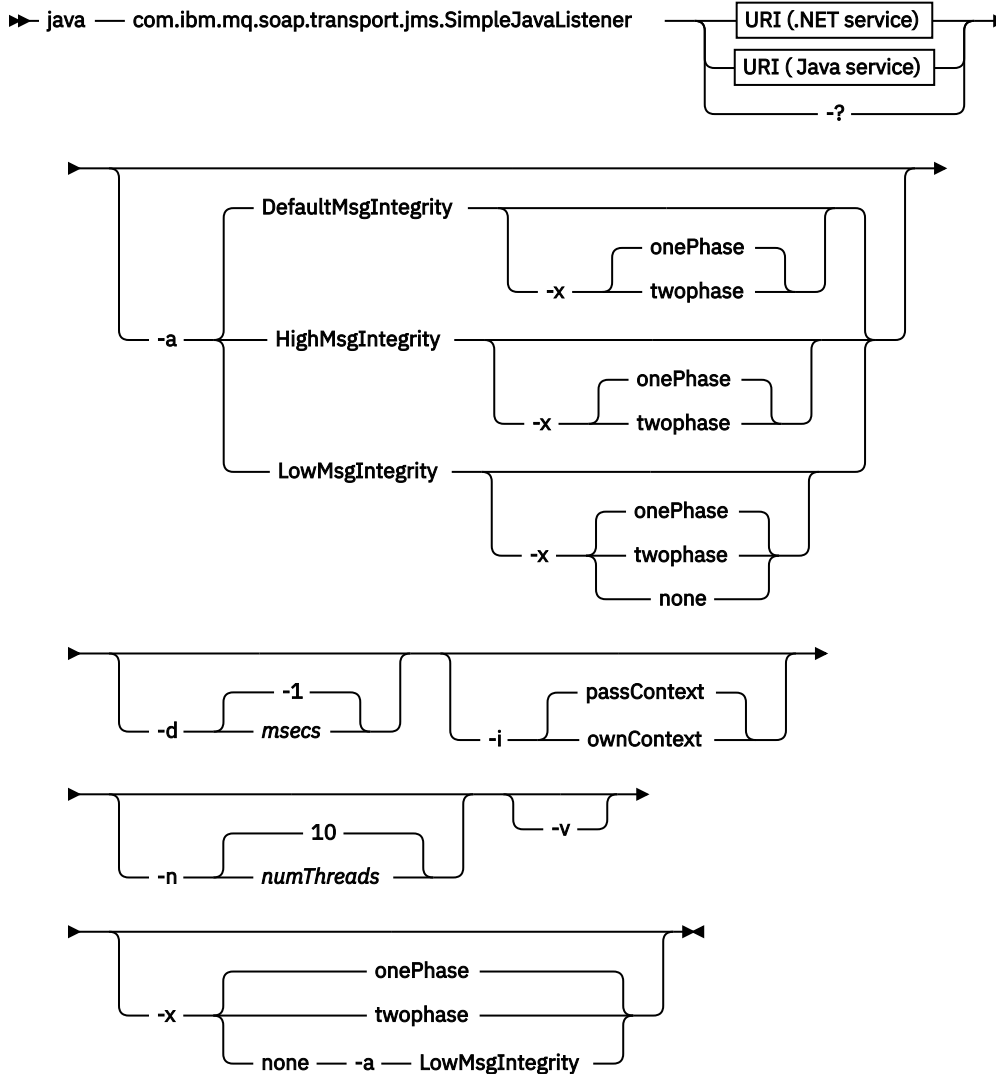
SimpleJavaListener: IBM MQ SOAP Listener for Axis 1.4

Syntax and parameters for the IBM MQ SOAP listener for Axis 1.4.

Purpose

Starts the IBM MQ SOAP listener for Axis 1.4.

Java



Required parameters

URI platform

See [“URI syntax and parameters for Web service deployment”](#) on page 1447.

-?

Print out help text describing how the command is used.

Optional parameters

-a integrityOption

integrityOption specifies the behavior of IBM MQ SOAP listeners if it is not possible to put a failed request message on the dead-letter queue. *integrityOption* can take one of the following values:

DefaultMsgIntegrity

For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. DefaultMsgIntegrity applies if the -a option is omitted, or if *integrityOption* is not specified.

LowMsgIntegrity

For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity

For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits.

The deployment utility checks for the compatibility of the -x and -a flags. If -x none is specified, then -a LowMsgIntegrity must be specified. If the flags are incompatible the deployment utility exits with an error message and with no deployment steps having been undertaken.

-d msecs

msecs specifies the number of milliseconds for the IBM MQ SOAP listener to stay alive if request messages have been received on any thread. If *msecs* is set to -1, the listener stays alive indefinitely.

-i Context

Context specifies whether the listeners pass identity context. *Context* takes the following values:

passContext

Set the identity context of the original request message into the response message. The SOAP listener checks that it has authority to save the context from the request queue and to pass it to the response queue. It makes the checks at run time when opening the request queue to save context, and the response queue to pass context. If it does not have the required authority, or the MQOPEN call fails, and the response message is not processed. The response message is put on the dead-letter queue with the dead-letter header containing the return code from the failed MQOPEN. The listener then continues to process subsequent incoming messages as normal.

ownContext

The SOAP listener does not pass context. The returned context reflects the user ID under which the listener is running rather than the user ID which created the original request message.

The fields in the origin context are set by the queue manager, and not by the SOAP listener.

-n numThreads

numThreads specifies the number of threads in the generated startup scripts for the IBM MQ SOAP listener. The default is 10. Consider increasing this number if you have high message throughput.

-v

-v sets verbose output from external commands. Error messages are always displayed. Use -v to output commands that you can tailor to create customized deployment scripts.

-w serviceDirectory

serviceDirectory is the directory containing the web service.

-x transactionality

transactionality specifies the type of transactional control for the listener. *transactionality* can be set to one of the following values:

onePhase

IBM MQ one-phase support is used. If the system fails during processing, the request message is redelivered to the application. IBM MQ transactions ensure that the response messages are written exactly once.

twoPhase

Two-phase support is used. If the service is written appropriately the message is delivered exactly once, coordinated with other resources, within a single committed execution of the service. This option applies to server bindings connections only.

none

No transactional support. If the system fails during processing, the request message can be lost, even if persistent. The service might or might not have executed, and response, report or dead-letter messages might or might not be written.

The deployment utility checks for the compatibility of the -x and -a flags. See the description of the -a flag for details.

Java Example

```
java com.ibm.mq.soap.transport.jms.SimpleJavaListener
-u "jms:/queue?destination=myQ&connectionFactory=()
```

```
&initialContextFactory=com.ibm.mq.jms.NoJndi"
-n 20
```

IBM MQ SOAP listeners

An IBM MQ SOAP listener reads an incoming SOAP request from the queue specified as the destination in the URI. It checks the format of the request message and then invokes a Web service using the Web services infrastructure. An IBM MQ SOAP listener returns any response or error from a Web service using the reply destination queue in the URI. It returns IBM MQ reports to the reply queue.

The term listener is used here in its standard Web services sense. It is distinct from the standard IBM MQ listener invoked by the **runmqcls** command.

Description

The Java SOAP listener is implemented as a Java class and run services using Axis 1.4. The .NET listener is a console application and runs .NET Framework 1 or .NET Framework 2 services. For .NET Framework 3 services, use the IBM MQ custom channel for Microsoft Windows Communication Foundation (WCF).

The deployment utility creates scripts to start Java or .NET SOAP listeners automatically. A SOAP Listener can be started manually using either the **amqSOAPNETListener** command, or by calling the `SimpleJavaListener` class. You can configure the IBM MQ SOAP listener to be started as an IBM MQ service by setting the **-s** option in the deployment utility. Alternatively, start listeners using triggering, or use the start and end listener scripts generated by the deployment utility. You can configure triggering manually, or use the **-tmq** and **-tmp** deployment options to configure triggering automatically. You can end a listener by setting the request queue to GET (DISABLED).

Table 636. Command scripts generated by the deployment utility			
Web service Infrastructure	UNIX and Linux systems	Windows Java	Windows.NET
Start listener	<code>startWMQJListener.sh</code>	<code>startWMQJListener.cmd</code>	<code>startWMQNListener.cmd</code>
Stop listener	<code>endWMQJListener.sh</code>	<code>endWMQJListener.cmd</code>	<code>endWMQNListener.cmd</code>
Define listener service	<code>defineWMQJListener.sh</code>	<code>defineWMQJListener.cmd</code>	<code>defineWMQNListener.cmd</code>

The IBM MQ SOAP listener passes the `endpointURL` and `soapAction` fields from the SOAP message to the SOAP infrastructure. The listener invokes the service through the Web Services infrastructure and waits for the response. The listener does not validate `endpointURL` and `soapAction`. The fields are set by the IBM MQ SOAP sender from the data that is provided in the URI set by a SOAP client.

The listener creates the response message and sends it to the reply destination supplied in the request message URI. In addition, the listener sets the correlation ID in the response message according to the report option in the request message. It returns the expiry, persistence, and priority settings from the request message. The listener also sends report messages back to clients in some circumstances.

If there are format errors in the SOAP request, the listener returns a report message to the client using the reply destination queue. The queue manager also returns report messages to the client using the reply destination queue, if a report has been requested. Full report messages are written to the response queue, in response to a number of events:

- An exception.
- Message expiry.
- Format of the request message is not recognized.
- Failure of the integrity check of the **MQRFH2** header.
- The format of the main message body is not **MQFMT_NONE**.

- The backout/retry threshold is exceeded while the IBM MQ SOAP listener is processing the request.

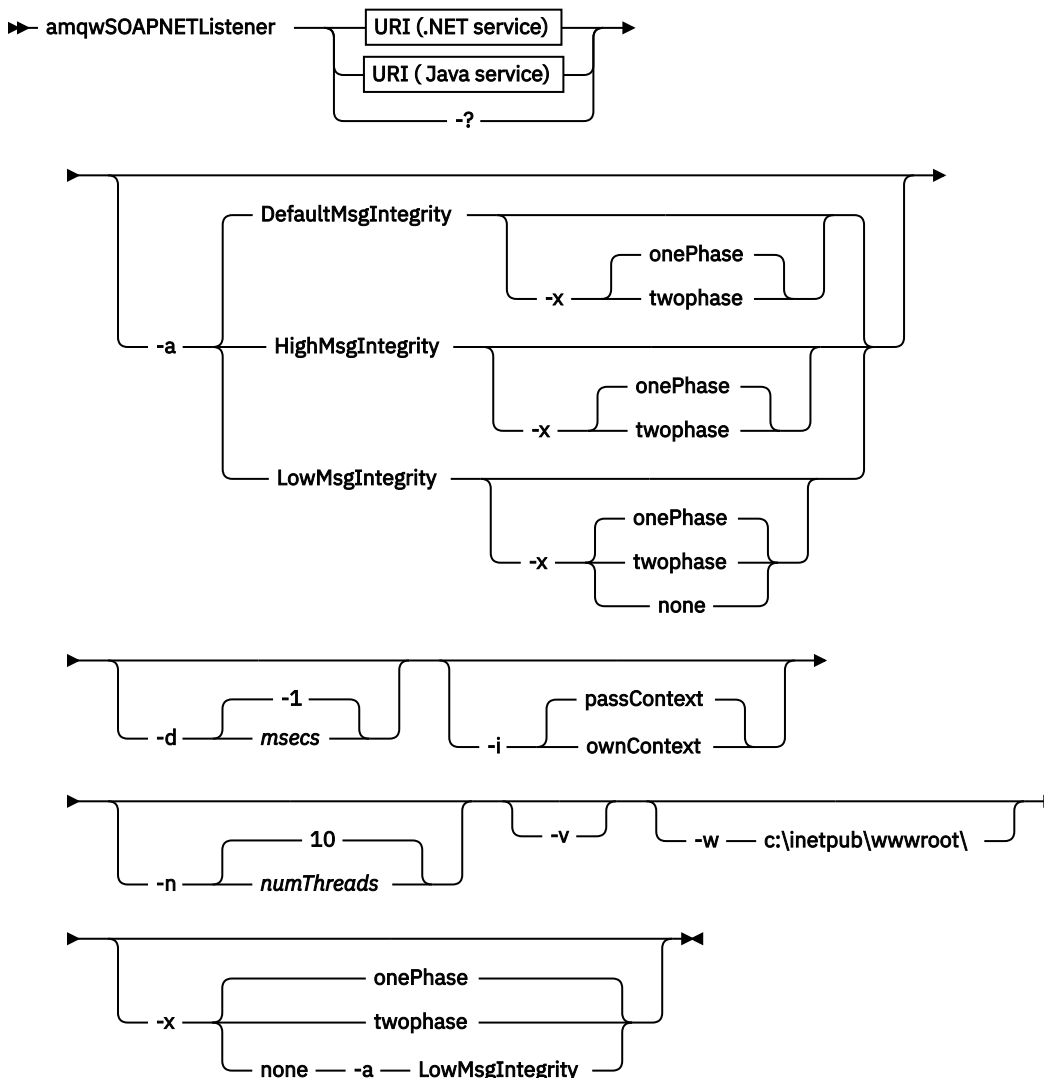
The IBM MQ SOAP sender sets MQRO_EXCEPTION_WITH_FULL_DATA and MQRO_EXPIRATION_WITH_FULL_DATA report options. As a result of the report options set by the IBM MQ SOAP sender, the report message contains the entire originating request message. The IBM MQ SOAP sender also sets the MQRO_DISCARD option, which causes the message to be discarded after a report message has been returned. If the report options do not meet your requirements, write your own senders to use different MQRO_EXCEPTION and MQRO_DISCARD report options. If the SOAP request is sent by a different sender that did not set MQRO_DISCARD, the failing message is written to the dead letter queue (DLQ).

If the listener generates a report message but fails in the process of sending the report, the report message is sent to the DLQ. Ensure that your DLQ handler handles these messages correctly.

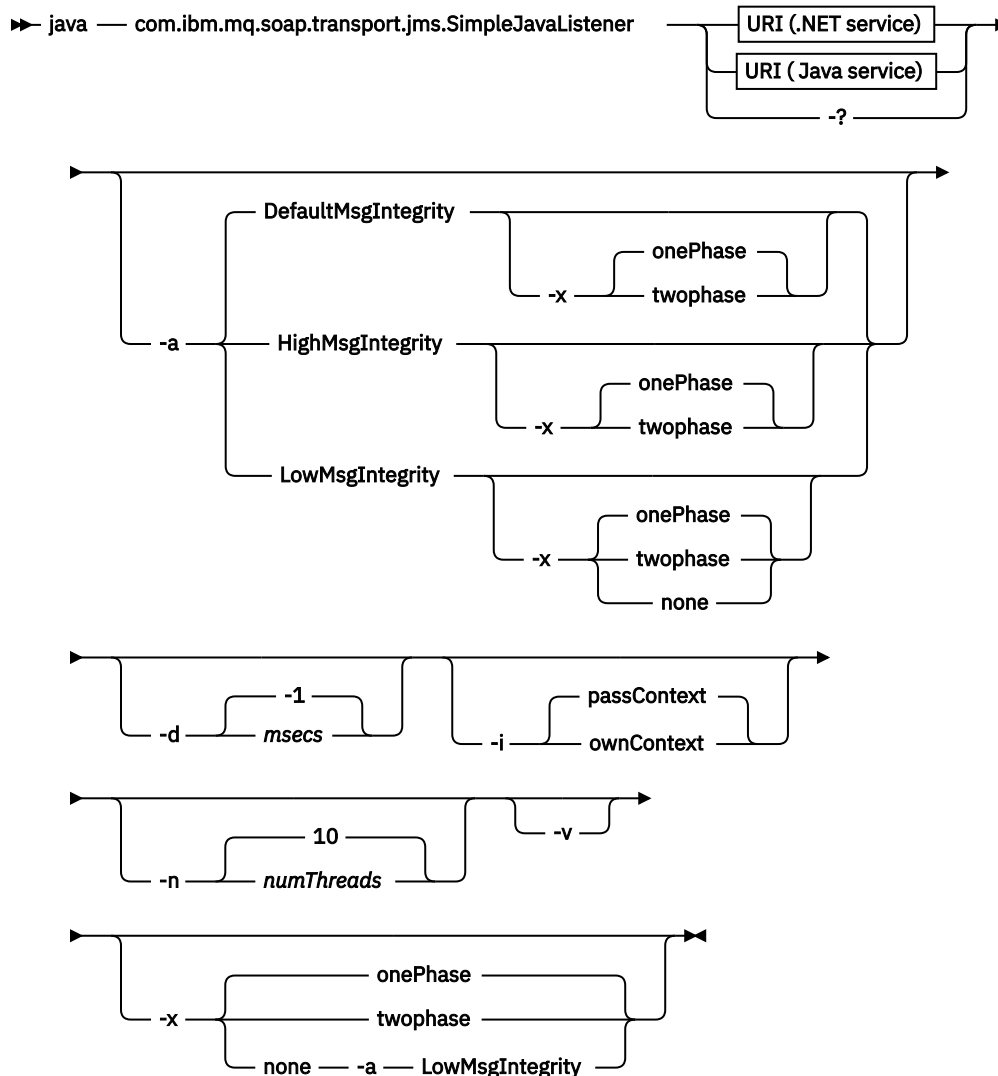
If an error occurs when attempting to write to the dead-letter queue a message is written to the IBM MQ error log. Whether the listener continues to process more messages depends on which message persistence and transactional options are selected. If the listener is running in one-phase transactional mode and is processing a non-persistent request message, the original message is discarded. The IBM MQ SOAP listener continues to execute. If the request message is persistent the request message is backed out to the request queue and the listener exits. The request queue is set to get-inhibited to prevent an accidental triggered restart.

Syntax diagram

.NET



Java



Required parameters

URI platform

See [“URI syntax and parameters for Web service deployment”](#) on page 1447.

-?

Print out help text describing how the command is used.

Optional parameters

-a integrityOption

integrityOption specifies the behavior of IBM MQ SOAP listeners if it is not possible to put a failed request message on the dead-letter queue. *integrityOption* can take one of the following values:

DefaultMsgIntegrity

For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. DefaultMsgIntegrity applies if the -a option is omitted, or if *integrityOption* is not specified.

LowMsgIntegrity

For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity

For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits.

The deployment utility checks for the compatibility of the -x and -a flags. If -x none is specified, then -a LowMsgIntegrity must be specified. If the flags are incompatible the deployment utility exits with an error message and with no deployment steps having been undertaken.

-d msec

msec specifies the number of milliseconds for the IBM MQ SOAP listener to stay alive if request messages have been received on any thread. If *msec* is set to -1, the listener stays alive indefinitely.

-i Context

Context specifies whether the listeners pass identity context. *Context* takes the following values:

passContext

Set the identity context of the original request message into the response message. The SOAP listener checks that it has authority to save the context from the request queue and to pass it to the response queue. It makes the checks at run time when opening the request queue to save context, and the response queue to pass context. If it does not have the required authority, or the MQOPEN call fails, and the response message is not processed. The response message is put on the dead-letter queue with the dead-letter header containing the return code from the failed MQOPEN. The listener then continues to process subsequent incoming messages as normal.

ownContext

The SOAP listener does not pass context. The returned context reflects the user ID under which the listener is running rather than the user ID which created the original request message.

The fields in the origin context are set by the queue manager, and not by the SOAP listener.

-n numThreads

numThreads specifies the number of threads in the generated startup scripts for the IBM MQ SOAP listener. The default is 10. Consider increasing this number if you have high message throughput.

-v

-v sets verbose output from external commands. Error messages are always displayed. Use -v to output commands that you can tailor to create customized deployment scripts.

-w serviceDirectory

serviceDirectory is the directory containing the web service.

-x transactionality

transactionality specifies the type of transactional control for the listener. *transactionality* can be set to one of the following values:

onePhase

IBM MQ one-phase support is used. If the system fails during processing, the request message is redelivered to the application. IBM MQ transactions ensure that the response messages are written exactly once.

twoPhase

Two-phase support is used. If the service is written appropriately the message is delivered exactly once, coordinated with other resources, within a single committed execution of the service. This option applies to server bindings connections only.

none

No transactional support. If the system fails during processing, the request message can be lost, even if persistent. The service might or might not have executed, and response, report or dead-letter messages might or might not be written.

The deployment utility checks for the compatibility of the -x and -a flags. See the description of the -a flag for details.

.NET Example

```
amqwSOAPNETlistener  
-u "jms:/queue?destination=myQ&connectionFactory=()
```

```
&targetService=myService&initialContextFactory=com.ibm.mq.jms.NoJndi"  
-w C:/wmqsoap/demos  
-n 20
```

Java Example

```
java com.ibm.mq.soap.transport.jms.SimpleJavaListener  
-u "jms:/queue?destination=myQ&connectionFactory=()  
&initialContextFactory=com.ibm.mq.jms.NoJndi"  
-n 20
```

IBM MQ transport for SOAP sender

Sender classes are provided for Axis and .NET Framework 1 and .NET Framework 2. The sender constructs a SOAP request and puts it on a queue, it then blocks until it has read a response from the response queue. You can alter the behavior of the classes by passing different URIs from a SOAP client. For .NET Framework 3 use IBM MQ custom channel for Microsoft Windows Communication Foundation (WCF).

Purpose

The IBM MQ SOAP sender puts a SOAP request to invoke a Web service onto an IBM MQ request queue. The sender sets fields in the **MQRFH2** header according to options specified in the URI, or according to defaults.

If you need to change the behavior of a sender beyond what is possible using the URI options, write your own sender. Your sender can work with the IBM MQ transport for SOAP listeners, or with other SOAP environments. Your sender must construct SOAP messages in the format defined by IBM MQ. The format is supported by the IBM MQ SOAP listener, and also SOAP listeners provided by WebSphere Application Server and CICS. Your sender must follow the rules for an IBM MQ requestor. The IBM MQ SOAP listener returns reply and report messages. See [“MQMD SOAP settings” on page 1425](#) for details how to set the report options in the **MQMD**. The report options control the report messages returned by the IBM MQ SOAP listener.

Description

The IBM MQ SOAP Java sender is registered with the Axis host environment for the `jms:` URI prefix. The sender is implemented in the class `com.ibm.mq.soap.transport.jms.WMQSender`, which is derived from `org.apache.axis.handlers.BasicHandler`. If the Axis host environment detects a `jms:` URI prefix it invokes the `com.ibm.mq.soap.transport.jms.WMQSender` class. The class blocks after placing the message until it has read a response from the response queue. If no response is received within a timeout interval the sender throws an exception. If a response is received within the timeout interval the response message is returned to the client using the Axis framework. Your client application must be able to handle these response messages.

For Microsoft .NET Framework 1 and .NET Framework 2 services, the IBM MQ SOAP sender is implemented in the class `IBM.WMQSOAP.MQWebRequest`, which is derived from `System.Net.WebRequest` and `System.Net.IWebRequestCreate`. If .NET Framework 1 or .NET Framework 2 detects a `jms:` URI prefix it invokes the `IBM.WMQSOAP.MQWebRequest` class. The sender creates an `MQWebResponse` object to read the response message from the response queue and return it to the client.

`com.ibm.mq.soap.transport.jms.WMQSender` is a final class, and `IBM.WMQSOAP.MQWebRequest` is sealed. You cannot modify their behavior by creating subclasses.

Parameters

Set the URI to control the behavior of the IBM MQ SOAP sender in a Web service SOAP client. The deployment utility creates Web service client stubs incorporating the URI options supplied to the deployment utility.

Use a channel definition table with the IBM MQ SOAP transport for SOAP sender

A client connection channel definition is an alternative to setting connection properties in the `ConnectionFactory` attribute of the Web service URI. The connection properties are `clientChannel`, `clientConnection`, and `SSL` parameters.

Description

Create the client channel description table by defining client connections. Even if a Web services client connects to different queue managers, create all the connections in the connection table on a single queue manager. The default name and location of the connection table is *queue manager directory/@ipcc/AMQCLCHL.TAB*.

Pass the location of the connection table to a Java client by setting the `com.ibm.mq.soap.transport.jms.mqchlurl` system property.

Pass the location of the connection table to a .NET client by setting the `MQCHLLIB` and `MQCHLTAB` environment variables.

You might provide both a channel connection table and channel connection parameters in the `ConnectionFactory` attribute of the Web service URI. The values set in the `ConnectionFactory` take precedence over the values in the channel definition table.

Using a channel definition table in Java

```
java -Dcom.ibm.msg.client.config.location=file:/C:/mydir/myjms.config MyAppClass
```

Figure 21. Starting Java client using a configuration file

```
com.ibm.mq.soap.transport.jms.mqchlurl=file:/C:/ibm/wmq/qmgrs/QM1/@ipcc/AMQCLCHL.TAB
```

Figure 22. *myjms.config*

Transactions

Use the `-x` option when starting the listener, to run Web services transactionally. Select the integrity of messages by setting the `persistence` option in the service URI.

Web services

Use the `-x` option when starting the listener, to run Web services transactionally. On .NET Framework 1 and 2 the SOAP listener uses Microsoft Transaction Coordinator (MTS). On Axis 1.4, the SOAP listener uses queue manager coordinated transactions.

Web service clients

The SOAP senders are not transactional.

IBM MQ bindings

You can set the binding type for the SOAP sender. It can connect as an IBM MQ server application, or as a client application. You can also bind the SOAP sender as an XA-client on .NET.

Message persistence

Select the level of persistence by setting the `Persistence` option in the URI.

Web service transactions

You can use Web service transactions, because the SOAP sender is not transactional. If you write your own SOAP sender, and intend to use Web service transactions, do not create a transactional SOAP sender.

You cannot send the request message and receive the reply message in the same transaction. The send and receive must not be coordinated by the Web service transaction.

URI syntax and parameters for Web service deployment

The syntax and parameters to deploy an IBM MQ Web service are defined in a URI. The deployment utility generates a default URI based on the name of the Web service. You can override the defaults by defining your own URI as a parameter to the deployment utility. The deployment utility incorporates the URI in the generated Web service client stubs.

Purpose

A web service is specified using a Universal Resource Identifier (URI). The syntax diagram specifies the URI that is supported in the IBM MQ transport for SOAP. The URI controls over IBM MQ -specific SOAP parameters and options used to access target services. The URI is compatible with Web services hosted by .NET, Apache Axis 1, WebSphere Application Server, CICS.

Description

The URI is incorporated into the Web service client classes generated by the deployment utility. The client passes the URI to IBM MQ SOAP Sender in an IBM MQ message. The URI controls the processing performed by both the IBM MQ SOAP Sender and IBM MQ SOAP listener.

Syntax

The URI syntax is as follows:

```
jms:/queue?name=value&name=value...
```

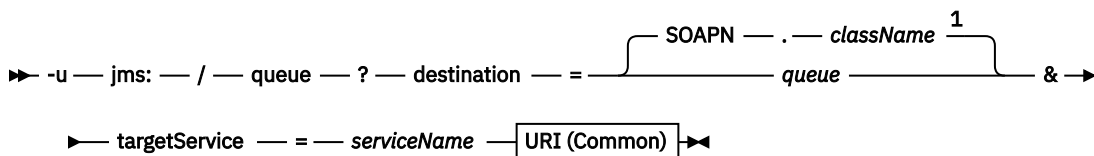
where name is a parameter name and *value* is an appropriate value, and the name = *value* element can be repeated any number of times with the second and subsequent occurrences being preceded by an ampersand (&).

Parameter names are case-sensitive, as are names of IBM MQ objects. If any parameter is specified more than once, the final occurrence of the parameter takes effect. Client applications can override a generated parameter by appending another copy of the parameter to the URI. If any additional unrecognized parameters are included, they are ignored.

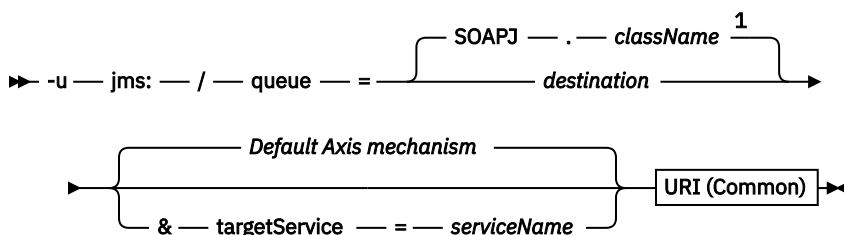
If you store a URI in an XML string, you must represent the ampersand character as & . Similarly, if a URI is coded in a script, take care to escape characters such as & that would otherwise be interpreted by the shell.

Syntax diagram

URI (.NET service)

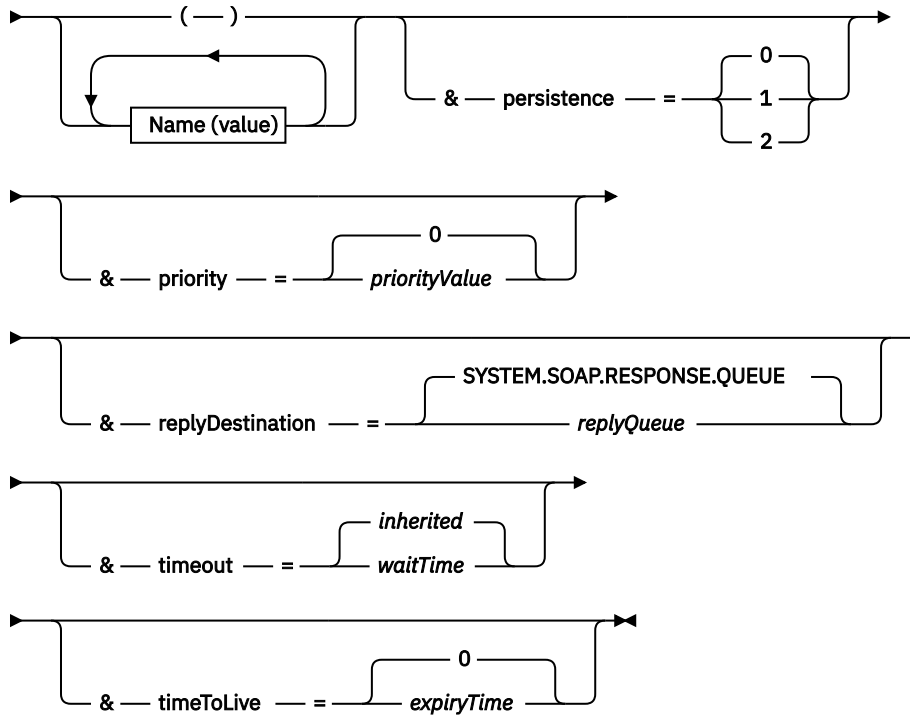


URI (Java service)

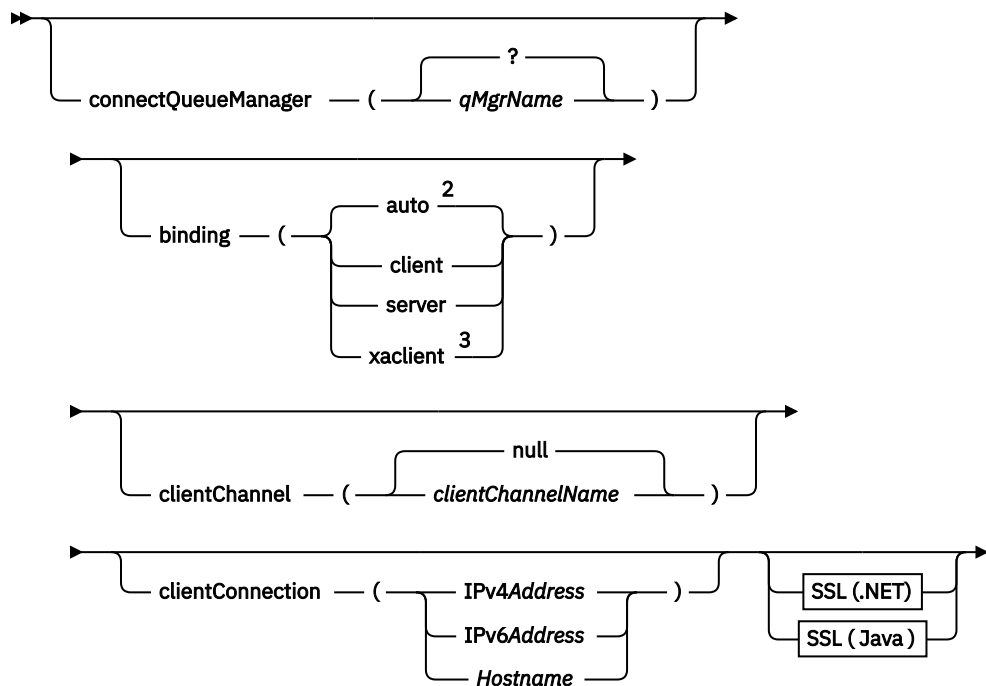


URI (Common)

►► & — initialContextFactory — = — com.ibm.mq.jms.Nojndi — & — connectionFactory — = —►



Name (value)



Notes:

- ¹ The queue manager transforms *className* to a queue name following the steps described in [“Destination to queue name transformation”](#) on page 1449
- ² `client` is the default if other options appropriate for a client are specified; for example `clientConnection`.
- ³ `xacient` applies to .NET only

Destination to queue name transformation

1. *className* is prefixed with SOAPJ . for Java services or with SOAPN . for .NET services.
2. The file extension is removed from the full path name given in the *className* parameter.
3. The resulting string is truncated to no more than 48 characters
4. Directory separator characters are replaced with period characters.
5. Embedded spaces are replaced with underscore characters.
6. The colon following a drive prefix letter is replaced with a period for a .NET service.

Note: In some environments, a queue name generated by the deployment utility might not be unique. The deployment utility makes checks whether to create the queue. You might choose to override the deployment utility by restructuring the deployment directory hierarchy, or by customizing the supplied deployment process.

Required URI parameters

destination = queue

queue is the name of the request destination. It can be a queue or a queue alias. If it is queue alias, the alias can resolve to a topic.

- If the -u parameter is omitted *queue* is generated from *classname* using the steps described in [“Destination to queue name transformation” on page 1449](#).
- If the -u parameter is specified *queue* is required and must be the first parameter of the URI after the initial `jms: / queue ?` string. Specify either an IBM MQ queue name, or a queue name and queue manager name connected by an @ symbol, for example SOAPN.trandemos@WMQSOAP.DEMO.QM.
- The deployment utility checks whether the queue name, generated or provided, matches the name of an existing queue. The action taken is described in [Table 637 on page 1449](#).

Table 637. queue validation			
Listener script exists?	Listener script exists in the ./generated/server directory		Listener script does not exist in the ./generated/server directory
Queue in listener script matches queue ?	queue does not match request queue being used in listener script	queue matches request queue being used in listener script	
queue exists	<ul style="list-style-type: none"> – Deployment exits with an error. – The service has already been deployed in ./generated/server, but using a different queue. 	<ul style="list-style-type: none"> – Deployment continues normally. – The service has already been deployed in ./generated/server 	<ul style="list-style-type: none"> – Deployment exits with an error. – The listener startup script is not found in ./generated/server, but queue is in use by a different service or application.
queue does not exist		<ul style="list-style-type: none"> – Deployment continues with a warning. – The previous deployment might have failed, because the startup is valid, but queue is missing. 	<ul style="list-style-type: none"> – Deployment continues normally. – No service has been deployed from this directory.

&connectionFactory = *Name(value)*

Name is one of the following parameters:

- [connectQueueManager\(qMgrName\)](#)
- [binding\(bindingType\)](#)
- [clientChannel\(channel\)](#)
- [clientConnection\(connection\)](#)
- [“Required SSL parameters \(Java \)” on page 1437](#)

See [“Connection factory parameters” on page 1451](#) for a description of the values of these parameters.

&targetService = *serviceName*

⁹ On .NET, *serviceName* is the name of a .NET service located in the deployment directory, for example: `targetService=myService.asmx`. In the .NET environment, the `targetService` parameter makes it possible for a single IBM MQ SOAP listener to be able to process requests for multiple services. These services must be deployed from the same directory.

Optional URI parameters**&initialContextFactory = *contextFactory***

contextFactory is required and must be set to `com.ibm.mq.jms.NoJndi`. Make sure `NoJndi.jar` is in the class path for a WebSphere Application Server Web services client. `NoJndi.jar` returns Java objects based on the contents of the `connectionFactory` and `destination` parameters, rather than by reference to a directory.

&targetService = *serviceName*

¹⁰ On Axis, *serviceName* is the fully qualified name of a Java service, for example: `targetService=javaDemos.service.StockQuoteAxis`. If `targetService` is not specified, a service is loaded using the default Axis mechanism.

&persistence = *messagePersistence*

messagePersistence takes one of the following values:

- 0**
Persistence is inherited from the queue definition.
- 1**
The message is non-persistent.
- 2**
The message is persistent

&priority = *priorityValue*

priorityValue is in the range 0 - 9. 0 is low priority. The default value is environment-specific, which in the case of IBM MQ is 0.

&replyDestination = *replyToQueue*

The queue at the client side to be used for the response message. The default reply queue is `SYSTEM.SOAP.RESPONSE.QUEUE`.

- Run the `setupWMQSOAP` script to create the default IBM MQ SOAP objects.
- Specify a model queue for *replyToQueue* to create either a temporary or permanent dynamic response queue. For both temporary and permanent dynamic response queues, a separate instance of dynamic queue is created for each request. If any of the following events happen the queue is deleted:
 - The response arrives and is processed.
 - The request times out.

⁹ .NET service only

¹⁰ Java service only

- The requesting program terminates.

For the best performance, use temporary dynamic queues rather than permanent dynamic queues. Do not send a persistent request message to a URI with a temporary dynamic queue. The IBM MQ listener SOAP fails to process the message and outputs an error. The client times out waiting for the reply.

- The `setupWMQSOAP` script creates a default permanent dynamic model queue called `SYSTEM.SOAP.MODEL.RESPONSE.QUEUE`.

&timeout = *waitTime*

The time, in milliseconds, that the client waits for a response message. *waitTime* overrides values set by the infrastructure or client application. If not specified the application value, if specified, or infrastructure default is inherited.

Note: No relationship is enforced between timeout and timeToLive.

&timeToLive = *expiryTime*

expiryTime is the time, specified in milliseconds, before the message expires. The default is zero, which indicates an unlimited lifetime.

Note: No relationship is enforced between timeout and timeToLive.

Connection factory parameters

connectQueueManager (*qMgrName*)

qMgrName specifies the queue manager to which the client connects. The default is blank.

binding (*bindingType*)

bindingType specifies how the client is connected to *qMgrName*. The default is `auto`. *bindingType* takes the following values:

auto

The sender tries the following connection types, in order:

1. If other options appropriate to a client connection are specified, the sender uses a client binding. The other options are `clientConnection` or `clientChannel`.
2. Use a server connection.
3. Use a client connection.

Use `binding (auto)` in the *URI* if there is no local queue manager at the SOAP client. A client connection is built for the SOAP client.

client

Use `binding (client)` in the *URI* to build a client configuration for the SOAP sender.

server

Use `binding (server)` in the *URI* to build a server configuration for the SOAP sender. If the connection has client type parameters the connection fails and an error is displayed by the IBM MQ SOAP sender. Client type parameters are `clientConnection`, `clientChannel`, or SSL parameters.

xaclient

`xaclient` is applicable only on .NET and not for Java clients. Use an XA-client connection.

clientChannel (*channel*)

The SOAP client uses *channel* to make an IBM MQ client connection. *channel* must match the name of a server connection channel, unless channel auto definition is enabled at the server. `clientChannel` is a required parameter, unless you have provided a Client Connection Definition table (CCDT).

Provide a CCDT in Java by setting `com.ibm.mq.soap.transport.jms.mqchlurl`. In .NET set the `MQCHLLIB` and `MQCHLTAB` environment variables; see [“Use a channel definition table with the IBM MQ SOAP transport for SOAP sender” on page 1446](#).

clientConnection (connection)

The SOAP client uses *connection* to make an IBM MQ client connection. The default hostname is localhost, and default port is 1414. If *connection* is a TCP/IP address, it takes one of three formats, and can be suffixed with a port number.

JMS clients can either use the format : hostname:port or 'escape' the brackets using the format %X where X is the hexadecimal value that represents the bracket character in the code page of the URI. For example, in ASCII, %28 and %29 for (and) respectively.

.Net clients can use the brackets explicitly : hostname(port) or use the 'escaped' format.

IPv4 address

For example, 192.0.2.0.

IPv6 address

For example, 2001:DB8:0:0:0:0:0:0.

Host name

For example, www.example.com%281687%29, www.example.com:1687, or www.example.com(1687).

SSL platform

See [“Required SSL parameters \(Java \)” on page 1437](#)

Sample URIs

Note:

1. & in the URI is encoded as &
2. All the parameter listed previously are applicable to the clients.
3. Only **destination**, **connectionFactory** and **initialContextFactory** are applicable to the WCF service.

```
jms:/queue?  
destination=myQ&amp;connectionFactory=()&amp;initialContextFactory=com.ibm.mq.jms.Nojndi
```

Figure 23. URI for an Axis service, supplying only required parameters

```
jms:/queue?destination=myQ&amp;connectionFactory=()&amp;targetService=MyService.asmx  
&amp;initialContextFactory=com.ibm.mq.jms.Nojndi
```

Figure 24. URI for a .NET service, supplying only required parameters

```
jms:/queue?destination=myQ@myRQM&amp;connectionFactory=connectQueueManager(myconnQM)  
binding(client)clientChannel(myChannel)clientConnection(myConnection)  
&amp;initialContextFactory=com.ibm.mq.jms.Nojndi
```

Figure 25. URI for an Axis service, supplying some optional connectionFactory parameters

```
jms:/queue?destination=myQ@myRQM&amp;connectionFactory=connectQueueManager(myconnQM)  
binding(client)clientChannel(myChannel)clientConnection(myConnection)  
sslPeerName(CN=MQ Test 1,O=IBM,S=Hampshire,C=GB)  
&amp;initialContextFactory=com.ibm.mq.jms.Nojndi
```

Figure 26. URI for an Axis service, supplying the sslPeerName option of the connectionFactory parameter

The Nojndi mechanism

The Nojndi mechanism enables JMS programs, which use JNDI interfaces, to use the same URI as IBM MQ programs, which do not use JNDI.

You can use the IBM MQ transport for SOAP to invoke Web services on WebSphere Application Server. WebSphere Application Server SOAP over JMS looks up the JMS resources using JNDI. The Web service client might be running on .NET, or using Axis 1.4, to invoke the Web service and not using JNDI. To use

the same URL for the client and server, it must provide the same information whether the environment is using JNDI or not.

The URI passed to the IBM MQ transport for SOAP by a Web service client contains a specific IBM MQ queue manager and queue names. These names are parsed and used directly by IBM MQ SOAP support.

The `Nojndi` mechanism directs the `initialContextFactory` used by a JMS program to `com.ibm.mq.jms.Nojndi`. The `com.ibm.mq.jms.Nojndi` class is an implementation of the JNDI interface that returns the `connectionFactory` and destination from the URL as `ConnectionFactory` and `Queue` Java objects. If the JMS implementation is IBM MQ, `MQConnectionFactory` and `MQQueue` inherit from the `ConnectionFactory` and `Queue` classes.

By using the `Nojndi` mechanism, you are able to provide the same connection information to WebSphere Application server and .NET using the same URL.

W3C SOAP over JMS URI for the IBM MQ Axis 2 client

Define a W3C SOAP over JMS URI to call a Web service from an Axis 2 client using IBM MQ JMS as the SOAP transport. The Web service must be provided by a server that supports IBM MQ JMS and the W3C SOAP over JMS candidate recommendation for the SOAP/JMS binding.

Description

The W3C candidate recommendation defines the SOAP over JMS binding; [SOAP over Java Message Service 1.0](#). Also useful for its examples is [URI Scheme for Java\(tm\) Message Service 1.0](#)¹¹.

Use the syntax diagram to create W3C SOAP over JMS URIs that are syntactically correct, and are accepted by the IBM MQ Axis 2 client. It is limited to defining the URI that is accepted by the IBM MQ Axis 2 client. It is a subset of the W3C recommendation in two respects:

1. The `jms-variant` topic is not supported, and must not be specified in a URI passed to the IBM MQ Axis 2 client.
2. The following properties are omitted from the syntax diagram because they are JMS properties, and not part of the URI.
 - a. `bindingVersion`
 - b. `contentType`
 - c. `soapAction`
 - d. `requestURI`
 - e. `isFault`

The JMS properties are set by the Axis 2 client or the server.

The diagram extends the W3C recommendation by defining a custom parameter, `connectionFactory`. `connectionFactory` is used as an alternative to JNDI to specify how the Axis 2 client connects to a queue manager using a queue.

The IBM MQ Axis 2 client only accepts properties as part of the URI passed to the client by the client application or as environment variables. The IBM MQ Axis 2 client has no capability to process a WSDL document. The client application or a development tool might process the WSDL and create the URI to pass to the Axis 2 client. An IBM MQ Axis 2 client application cannot set the JMS message properties directly.

Syntax

¹¹ Look for *URI Scheme for JMS*, in the W3C specification references, for the latest draft.

In accordance with the W3C recommendation, all the parameters can be obtained from environment variables. The environment variable names are formed by prefacing the parameter name with soapjms_. The syntax is: soapjms_ *parameterName* ; for example,

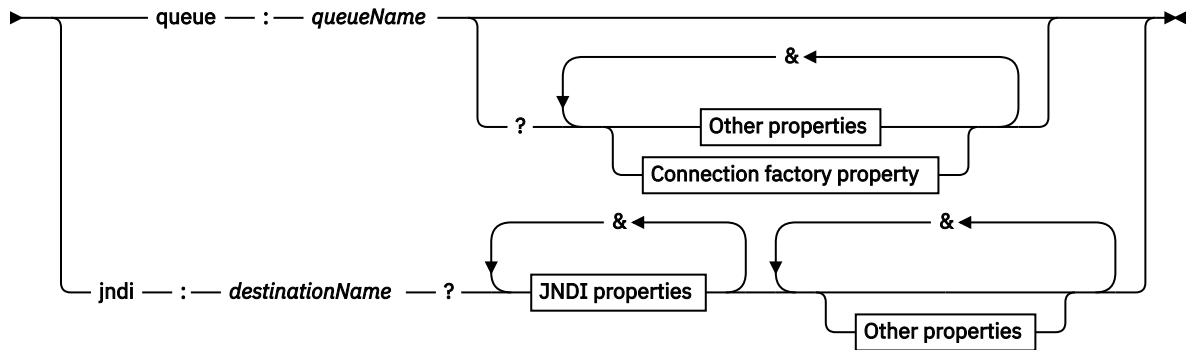
```
set soapjms_targetServer=com.example.org.stockquote
```

If a parameter is set using an environment variable it overrides the value set in the URI.

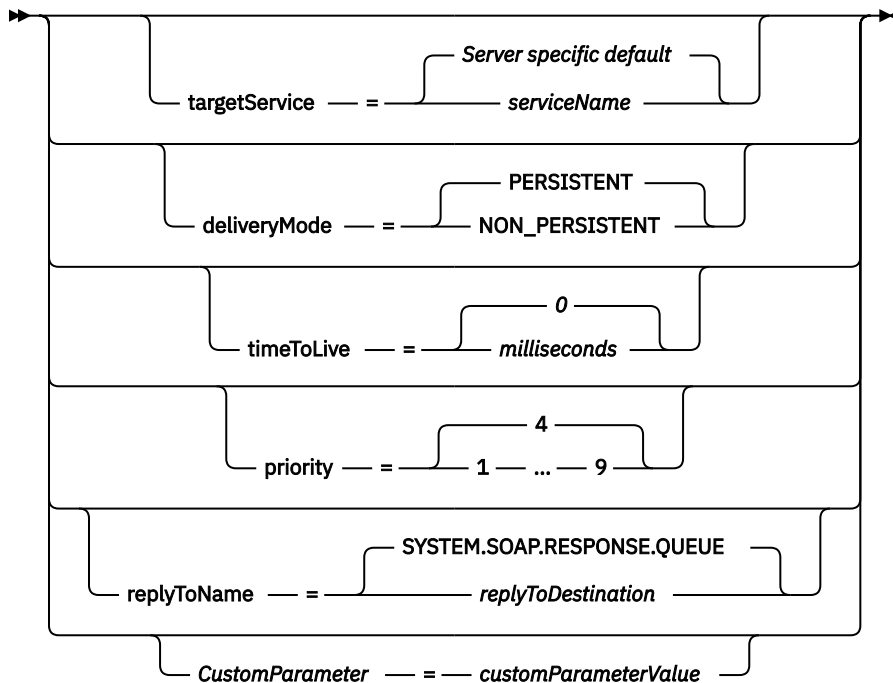
In accordance with the W3C recommendation, all the parameters can be repeated. The last instance of a parameter is used, unless overridden by an environment variable.

jms-uri

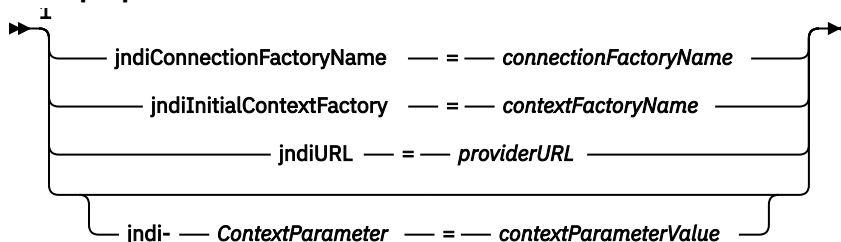
➔ jms: ➔



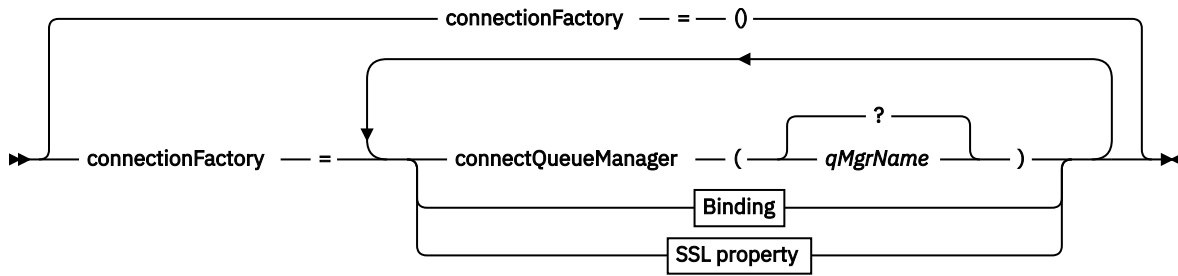
Other properties



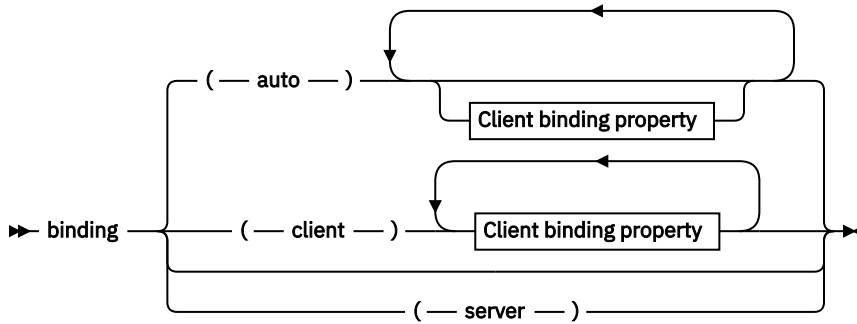
JNDI properties



Connection factory property

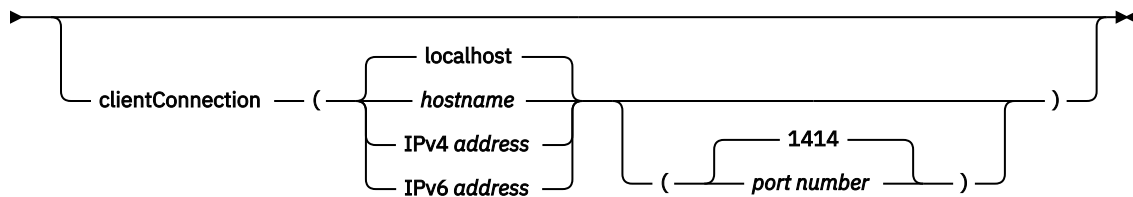


Binding



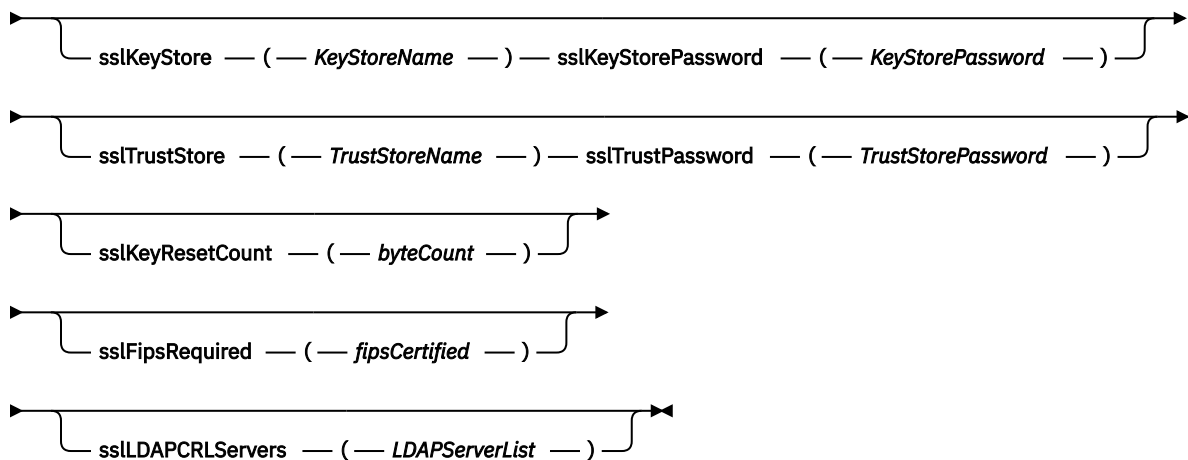
Client binding property

➤ `clientChannel` — (`channel`) ➔



SSL property

➤ `sslCipherSuite` — (`cipherSuite`) ➔



Notes:

¹ `jndiConnectionFactoryName`, `jndiConnectionFactoryName` and `jndiURL` are all required parameters. `jndi-ContextParameter` is optional.

Parameters

connectionFactory = *connectionFactoryParameterList*

connectionFactoryParameterList are parameters that qualify how the Axis 2 client connects to a queue manager when the destination variant is queue.

connectionFactory must not be specified with the jndi destination variant.

The parameters are not passed to the server in the request URI.

If connectionFactory is omitted, the queue must belong to a default queue manager running on the same server as the Axis 2 client.

The *connectionFactoryParameterList*:

binding (*bindingType*)

bindingType specifies how the client is connected to *qMgrName*. The default is auto. *bindingType* takes the following values:

auto

The sender tries the following connection types, in order:

1. If other options appropriate to a client connection are specified, the sender uses a client binding. The other options are clientConnection or clientChannel.
2. Use a server connection.
3. Use a client connection.

Use binding (auto) in the *URI* if there is no local queue manager at the SOAP client. A client connection is built for the SOAP client.

client

Use binding (client) in the *URI* to build a client configuration for the SOAP sender.

server

Use binding (server) in the *URI* to build a server configuration for the SOAP sender. If the connection has client type parameters the connection fails and an error is displayed by the IBM MQ SOAP sender. Client type parameters are clientConnection, clientChannel, or SSL parameters.

xaclient

xaclient is applicable only on .NET and not for Java clients. Use an XA-client connection.

clientChannel (*channel*)

The SOAP client uses *channel* to make an IBM MQ client connection. *channel* must match the name of a server connection channel, unless channel auto definition is enabled at the server. clientChannel is a required parameter, unless you have provided a Client Connection Definition table (CCDT).

Provide a CCDT in Java by setting `com.ibm.mq.soap.transport.jms.mqchlurl`. In .NET set the MQCHLLIB and MQCHLTAB environment variables; see [“Use a channel definition table with the IBM MQ SOAP transport for SOAP sender” on page 1446](#).

clientConnection (*connection*)

The SOAP client uses *connection* to make an IBM MQ client connection. The default hostname is localhost, and default port is 1414. If *connection* is a TCP/IP address, it takes one of three formats, and can be suffixed with a port number.

JMS clients can either use the format : hostname:port or 'escape' the brackets using the format %X where X is the hexadecimal value that represents the bracket character in the code page of the URI. For example, in ASCII, %28 and %29 for (and) respectively.

.Net clients can use the brackets explicitly : hostname(port) or use the 'escaped' format.

IPv4 address

For example, 192.0.2.0.

IPv6 address

For example, 2001:DB8:0:0:0:0:0:0.

Host name

For example, `www.example.com%281687%29`, `www.example.com:1687`, or `www.example.com(1687)`.

sslCipherSuite (*CipherSuite*)

CipherSuite specifies the sslCipherSuite used on the channel. The CipherSuite specified by the client must match the CipherSuite specified on the server connection channel.

sslFipsRequired (*fipsCertified*)

fipsCertified specifies whether *CipherSpec* or *CipherSuite* must use FIPS-certified cryptography in IBM MQ on the channel. The effect of setting *fipsCertified* is the same as setting the *FipsRequired* field of the **MQSCO** structure on an MQCONN call.

sslKeyStore (*KeyStoreName*)

KeyStoreName specifies the sslKeyStoreName used on the channel. The keystore holds the private key of the client used to authenticate the client to the server. The keystore is optional if the SSL connection is configured to accept anonymous client connections.

sslKeyResetCount (*bytecount*)

bytecount specifies the number of bytes passed across an SSL channel before the SSL secret key must be renegotiated. To disable the renegotiation of SSL keys omit the field or set it to zero. Zero is the only value supported in some environments, see [Renegotiating the secret key in IBM MQ classes for Java](#) . The effect of setting sslKeyResetCount is the same as setting the KeyResetCount field in the **MQSCO** structure on an MQCONN call.

sslKeyStorePassword (*KeyStorePassword*)

KeyStorePassword specifies the sslKeyStorePassword used on the channel.

sslLDAPCRLServers (*LDAPServerList*)

LDAPServerList specifies a list of LDAP servers to be used for Certificate Revocation List checking.

For SSL enabled client connections, *LDAPServerList* is a list of LDAP servers to be used for Certificate Revocation List (CRL) checking. The certificate provided by the queue manager is checked against one of the listed LDAP CRL servers; if found, the connection fails. Each LDAP server is tried in turn until connectivity is established to one of them. If it is impossible to connect to any of the servers, the certificate is rejected. Once a connection has been successfully established to one of them, the certificate is accepted or rejected depending on the CRLs present on that LDAP server.

If *LDAPServerList* is blank, the certificate belonging to the queue manager is not checked against a Certificate Revocation List. An error message is displayed if the supplied list of LDAP URIs is not valid. The effect of setting this field is the same as that of including MQAIR records and accessing them from an **MQSCO** structure on an MQCONN.

sslPeerName (*peerName*)

peerName specifies the sslPeerName used on the channel.

sslTrustStore (*TrustStoreName*)

TrustStoreName specifies the sslTrustStoreName used on the channel. The truststore holds the public certificate of the server, or its key chain, to authenticate the server to the client. The truststore is optional if the root certificate of a certificate authority is used to authenticate the server. In Java, root certificates are held in the JRE certificate store, `cacerts`.

sslTrustStorePassword (*TrustStorePassword*)

TrustStorePassword specifies the sslTrustStorePassword used on the channel.

CustomParameter = customParameterValue

CustomParameter is the user-defined name of a custom parameter, and *customParameterValue* is the value of the parameter.

Custom parameters that are not used by the Axis 2 client are sent by the Axis 2 client to the SOAP server. Consult the server documentation. *connectionFactory* is a custom parameter that is used by the Axis 2 client and is not passed to the server.

CustomParameter must not match the name of an existing parameter.

If *CustomParameter* starts with the string *jndi* - it is used in looking up a JNDI destination; see [jndi](#).

deliveryMode = *deliveryMode*

deliveryMode sets the message persistence. The default is PERSISTENT.

jndi : *destinationName*

destinationName is a JNDI destination name that maps to a JMS queue. If the *jndi* destination variant is specified, you must provide a *destinationName*.

jndiConnectionFactoryName = *connectionFactoryName*

connectionFactoryName sets the JNDI name of the connection factory. If the destination variant is *jndi*, *connectionFactoryName* must be provided.

jndiInitialContextFactory = *contextFactoryName*

contextFactoryName sets the JNDI name of the initial context factory. If the destination variant is *jndi*, *contextFactoryName* must be provided. See [Using JNDI to retrieve administered objects in a JMS application](#).

jndiURL = *providerURL*

jndiURL sets the URL name of the JNDI provider. If the destination variant is *jndi*, *jndiURL* must be specified.

jndi- ContextParameter = *contextParameterValue*

jndi- ContextParameter is the user-defined name of a custom parameter that used to pass information to the JNDI provider. *contextParameterValue* is the information that is passed.

priority = *priorityValue*

priorityValue sets the JMS message priority. 0 is low, 9 is high. The default value is 4.

queue : *queueName*

queueName is the name of a JMS queue on which the SOAP request is placed. If the queue variant is specified, a queue name must be provided. If the queue does not belong to a default queue manager on the same server as the client, set the [connectionFactory](#) parameter.

replyToName = *replyToDestination*

replyToDestination sets the destination queue name. If the destination variant is *jndi*, the name is a JNDI name that must map to a queue. If the variant is *queue* the name is a JMS queue. The default value is SYSTEM.SOAP.RESPONSE.QUEUE.

targetService = *serviceName*

The name used by the SOAP server to start the target Web service.

On Axis, *serviceName* is the fully qualified name of a Java service, for example:

`targetService=www.example.org.StockQuote`. If *targetService* is not specified, a service is loaded using the default Axis mechanism.

timeToLive = *milliseconds*

Set *milliseconds* to the time before the message expires. The default, 0, is the message never expires.

Examples

```
jms:jndi:REQUESTQ
?jndiURL=file:/C:/JMSAdmin
&jndiInitialContextFactory=com.sun.jndi.fscontext.RefFSContextFactory
&jndiConnectionFactoryName=ConnectionFactory
&replyToName=RESPONSEQ
&deliveryMode(NON_PERSISTENT)
```

Figure 27. Use *jms:jndi* to send a SOAP/JMS request

```
jms:queue:SOAPJ.demos
?connectionFactory=connectQueueManager(QM1)
Bind(Client)
ClientChannel(SOAPClient)
ClientConnection(www.example.org(1418))
&deliveryMode(NON_PERSISTENT)
```

Figure 28. Use `jms:queue` to send a SOAP/JMS request

Supported Web services

Code that has been written to run as a Web service does not need to be modified to use the IBM MQ transport for SOAP. You do need to deploy services differently to run with the IBM MQ transport for SOAP rather than using HTTP.

Description

V 8.0.0.2 From IBM MQ 8.0.0, Fix Pack 2, the IBM MQ transport for SOAP is deprecated. For more information, see [Deprecations: IBM MQ transport SOAP client and HTTP Bridge](#).

The IBM MQ transport for SOAP provides a SOAP listener to run services for .NET Framework 1 and .NET 2, and for Axis 1.4. The IBM MQ custom channel for Microsoft Windows Communication Foundation runs services for .NET Framework 3. WebSphere Application Server and CICS provide support for running services over IBM MQ transport for SOAP. Create a custom Export to use WebSphere Enterprise Service Bus or WebSphere Process Server.

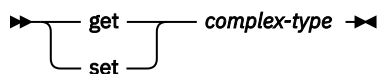
The IBM MQ SOAP listener can process SOAP requests transactionally. Run **amqwdployWMQService** using the `-x` option. The two-phase option is only supported for listeners using server bindings. Other environments might provide transactional support for the IBM MQ transport for SOAP. Consult their documentation.

IBM MQ transport for SOAP currently does not support the emerging industry standard SOAP over JMS protocol that has been submitted to W3C. You can distinguish a SOAP/JMS message written to the new standard by looking for the JMS BindingVersion property. IBM MQ transport for SOAP does not set the BindingVersion property.

Axis 1.4

A Java class can typically be used without modification. The types of any arguments to the methods in the web service must be supported by the Axis engine. Refer to Axis documentation for further details. If the service uses a complex object as an argument, or returns one, that object must comply to the Java bean specification. See the examples in [Figure 31 on page 1461](#), [Figure 32 on page 1461](#), and [Figure 33 on page 1461](#):

1. Have a public parameter-less constructor.
2. Any complex types of the bean must have public getters and setters of the form:



Prepare the service for deployment using the **amqwdployWMQService** utility. The service is invoked by the IBM MQ SOAP listener which uses `axis.jar` to run the service.

The only two-phase transaction manager supported for Axis 1.4 is IBM MQ.

The supplied deployment utility does not support the case where a service returns an object in a different package to the service itself. To use an object returned in a different package, write your own deployment utility. You can base your deployment utility on the supplied sample, or capture the commands it produces, using the `-v` option. Amend the commands to produce a tailored script.

If the service uses classes that are external to the Axis infrastructure and the IBM MQ SOAP run time environment, you must set the correct CLASSPATH. To change CLASSPATH, amend the generated script that starts or defines the listeners to include the services required, in one of the following ways:

- Amend the CLASSPATH directly in the script after the call to **amqwsetcp**.
- Create a service-specific script to customize the CLASSPATH and invoke this script in the generated script after the call to **amqwsetcp**.
- Create a customized deployment process to customize the CLASSPATH in the generated script automatically.

.NET Framework 1 and .NET Framework 2

A service that has already been prepared as an HTTP Web service does not need to be modified for use as an IBM MQ Web service. It needs to be deployed using the **amqwdeployWMQService** utility.

The only two-phase transaction manager supported for .NET Framework 1 and .NET 2 is Microsoft Transaction Server (MTS).

If the service code has not been prepared as an HTTP Web service you must convert it to a Web service. Declare the class as a web service and identify how the parameters of each method are formatted. You must check that any arguments to the methods of the service are compatible with the environment. Figure 29 on page 1461 and Figure 30 on page 1461 show a .NET class that has been prepared as a web service. The additions made are shown in bold type.

Figure 29 on page 1461 uses the code-behind programming model for a .NET Web service. In the code-behind model, the source for the service is separated from the .asmx file. The .asmx file declares the name of the associated source file with the Codebehind keyword. IBM MQ has samples of both inline and code-behind .NET Web services.

.NET Web services source must be compiled before deployment by the **amqwdeployWMQService** deployment utility. The service is compiled into a library (.dll). The library must be placed in the . /bin subdirectory of the deployment directory.

.NET Framework 3

Create an IBM MQ custom channel for Microsoft Windows Communication Foundation (WCF) to invoke services deployed to .NET Framework 3. See [Developing WCF applications with IBM MQ](#) for a description of how to configure WCF to use the IBM MQ transport for SOAP.

WebSphere Application Server

You can invoke web services hosted by WebSphere Application Server using the IBM MQ Transport for SOAP. See [Using SOAP over JMS to transport web services](#).

You need to modify the WSDL generated by deployment of a JMS service to WebSphere Application Server in order to generate a Web services client. The WSDL created by deployment to WebSphere Application Server includes a URI with a JNDI reference to the JMS InitialContextFactory. You need to modify the JNDI reference to Nojndi and provide connection attributes as described in [“URI syntax and parameters for Web service deployment” on page 1447](#).

CICS

You can invoke CICS applications using the IBM MQ Transport for SOAP. See [Configuring your CICS system for Web services](#).

WebSphere Enterprise Service Bus and WebSphere Process Server for Multiplatforms

WebSphere ESB and WebSphere Process Server for Multiplatforms support SOAP over JMS, with a ready built binding, only when using the default WebSphere Application Server messaging provider. Create a custom binding for JMS to support IBM MQ transport for SOAP. See [JMS data bindings](#).

Example

```
<%@ WebService Language="C#" CodeBehind="Quote.aspx.cs" Class="Quote.QuoteDotNet" %>
```

Figure 29. Service definition for .NET Framework 2: *Quote.aspx*

```
<%@ WebService Language="C#" CodeBehind="Quote.aspx.cs" Class="Quote.QuoteDotNet" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

namespace Quote {
    [WebService(Namespace = "http://www.example.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    public class QuoteDotNet : System.Web.Services.WebService {
        [WebMethod]
        public string getQuote(String symbol){
            return symbol.ToUpper();
        }
    }
}
```

Figure 30. Service implementation for .NET Framework 2: *Quote.aspx.cs*

```
package org.example.www;
public interface CustomerInfoInterface extends java.rmi.Remote {
    public org.example.www.CustomerRecord
        getCustomerName(org.example.www.CustomerRecord request)
        throws java.rmi.RemoteException, org.example.www.GetCustomerName_faultMsg;
}
```

Figure 31. Java JAX-RPC service interface using a complex type

```
package org.example.www;
public class CustomerInfoPortImpl implements org.example.www.CustomerInfoInterface{
    public org.example.www.CustomerRecord
        getCustomerName(org.example.www.CustomerRecord request)
        throws java.rmi.RemoteException, org.example.www.GetCustomerName_faultMsg {
        request.setName(request.getID().toString());
        return request;
    }
}
```

Figure 32. Java JAX-RPC service implementation using a complex type

```
package org.example.www;
public class CustomerRecord {
    private java.lang.String name;
    private java.lang.Integer ID;
    public CustomerRecord() {}
    public java.lang.String getName() {
        return name;
    }
    public void setName(java.lang.String name) {
        this.name = name;
    }
    public java.lang.Integer getID() {
        return ID;
    }
    public void setID(java.lang.Integer ID) {
        this.ID = ID;
    }
}
```

Figure 33. Java JAX-RPC service bean implementation of a complex type

IBM MQ transport for SOAP Web service clients

You can reuse an existing SOAP over HTTP client with IBM MQ transport for SOAP. You must make some small modifications to the code and build process to convert the client to work with IBM MQ transport for SOAP.

Coding

JAX-RPC clients must be written in Java. .NET Framework 1 and 2 clients can be written in any language that uses the Common Language Runtime. Code examples are provided in C# and Visual Basic.

The level of transactional support depends on the client environment and the pattern of the SOAP interaction. The SOAP request and SOAP reply can not be part of the same atomic transaction.

You must call `IBM.WMQSOAP.Register.Extension()` in a .NET Framework 1, .NET Framework 2 client. In a JAX-RPC Java Web service client call `com.ibm.mq.soap.Register.extension` to register the IBM MQ SOAP sender. The method registers the IBM MQ transport for SOAP sender as the handler for SOAP messages using the `jms: protocol`.

To create a .NET Framework 3 client, generate a Windows Communication Foundation client proxy using the **svcutil** tool; see [Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service](#).

Libraries required to build and run .NET Framework 1 and 2 clients

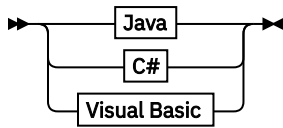
- `amqsoap`
- `System`
- `System.Web.Services`
- `System.Xml`

Libraries required to build and run Axis 1.4 clients

- `MQ_Install\java\lib\com.ibm.mq.soap.jar;`
- `MQ_Install\java\lib\com.ibm.mq.commonservices.jar;`
- `MQ_Install\java\lib\soap\axis.jar;`
- `MQ_Install\java\lib\soap\jaxrpc.jar`
- `MQ_Install\java\lib\soap\saa.jar;`
- `MQ_Install\java\lib\soap\commons-logging-1.0.4.jar;`
- `MQ_Install\java\lib\soap\commons-discovery-0.2.jar;`
- `MQ_Install\java\lib\soap\wsdl4j-1.5.1.jar;`
- `MQ_Install\java\jre\lib\xml.jar;`
- `MQ_Install\java\lib\soap\servlet.jar;`
- `MQ_Install\java\lib\com.ibm.mq.jar;`
- `MQ_Install\java\lib\com.ibm.mq.headers.jar;`
- `MQ_Install\java\lib\com.ibm.mq.pcf.jar;`
- `MQ_Install\java\lib\com.ibm.mq.jmqi.jar;`

Note: From IBM MQ 8.0, `ldap.jar`, `jndi.jar` and `jta.jar` have been removed from this list as they are now part of the JDK.

Register SOAP extension



Java

➔ `com.ibm.mq.soap.Register.extension()` ➔

C#

➔ `IBM.WMQSOAP.Register.Extension();` ➔

Visual Basic

➔ `IBM.WMQSOAP.Register.Extension` ➔

Client examples

Figure 34 on page 1463 is an example of a .NET Framework 1 or .NET Framework 2 C# client that uses the inline programming model. The **IBM.WMQSOAP.Register.Extension()** method registers the IBM MQ SOAP sender with .NET as the `jms:` protocol handler.

```
using System;
namespace QuoteClientProgram {
    class QuoteMain {
        static void Main(string[] args) {
            try {
                IBM.WMQSOAP.Register.Extension();
                Quote q = new Quote();
                Console.WriteLine("Response is: " + q.getQuote("ibm"));
            } catch (Exception e) {
                Console.WriteLine("Exception is: " + e);
            }
        }
    }
}
```

Figure 34. C# Web service client sample


Figure 35 on page 1463 is an example of a Java client that uses the JAX-RPC static proxy client interface. The **com.ibm.mq.soap.Register.extension();** method registers the IBM MQ SOAP sender with the service proxy to handle the `jms:` protocol.

```
package org.example.www;
import com.ibm.mq.soap.Register;
public class QuoteClient {
    public static void main(String[] args) {
        try {
            Register.extension();
            QuoteSOAPImplServiceLocator locator = new QuoteSOAPImplServiceLocator();
            System.out.println("Response = "
                + locator.getOrgExampleWwwQuoteSOAPImpl_Wmq().getQuote("IBM"));
        } catch (Exception e) {
            System.out.println("Exception = " + e.getMessage());
        }
    }
}
```

Figure 35. Java Web service client example

User exits, API exits, and installable services reference

Use the links provided in this section to help you develop your User exits, API exits, and installable services applications:

- [“MQIEP structure” on page 1464](#)
- [“Data-conversion exit reference” on page 1468](#)
- [“MQ_PUBLISH_EXIT - Publish exit” on page 1472](#)
- [“Channel-exit calls and data structures” on page 1480](#)
- [“API exit reference” on page 1566](#)
- [“Installable services interface reference information” on page 1624](#)
-  [“Installable services interface reference information” on page 1686](#)

Related concepts

[“MQI applications reference” on page 7](#)

Use the links provided in this section to help you develop your Message Queue Interface (MQI) applications.

[The IBM MQ Classes for Java libraries](#)

Related tasks

[Developing applications](#)

Related reference

[“SOAP reference” on page 1411](#)

[IBM MQ transport for SOAP reference information arranged alphabetically.](#)

[“Reference material for IBM MQ bridge for HTTP” on page 1725](#)

[Reference topics for IBM MQ bridge for HTTP, arranged alphabetically](#)

[“The IBM MQ .NET classes and interfaces” on page 1760](#)

IBM MQ .NET classes and interfaces are listed alphabetically. The properties, methods and constructors are described.

[“IBM MQ C++ classes” on page 1820](#)

The IBM MQ C++ classes encapsulate the IBM MQ Message Queue Interface (MQI). There is a single C++ header file, **imqi.hpp**, which covers all of these classes.

[IBM MQ Classes for JMS](#)

MQIEP structure

The MQIEP structure contains an entry point for each function call that exits are permitted to make.

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQIEP_STRUC_ID

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQIEP_VERSION_1

Version 1 structure version number.

MQIEP_CURRENT_VERSION

Current version of the structure.

StrucLength

Type: MQLONG

Size of the MQIEP structure in bytes. The value is as follows:

MQIEP_LENGTH_1**Flags**

Type: MQLONG

Provides information about the function addresses. A flag to indicate if the library is threaded can be used with a flag to indicate if the library is a client or server library.

The following value is used to specify no library information:

MQIEPF_NONE

One of the following values is used to specify if the shared library is threaded or non-threaded:

MQIEPF_NON_THREADED_LIBRARY

A non-threaded shared library

MQIEPF_THREADED_LIBRARY

A threaded shared library

One of the following values is used to specify if the shared library is a client or a server shared library:

MQIEPF_CLIENT_LIBRARY

A client shared library

MQIEPF_LOCAL_LIBRARY

A server shared library

Reserved

Type: MQPTR

MQBACK_Call

Type: PMQ_BACK_CALL

Address of the MQBACK call.

MQBEGIN_Call

Type: PMQ_BEGIN_CALL

Address of the MQBEGIN call.

MQBUFMH_Call

Type: PMQ_BUFMH_CALL

Address of the MQBUFMH call.

MQCB_Call

Type: PMQ_CB_CALL

Address of the MQCB call.

MQCLOSE_Call

Type: PMQ_CLOSE_CALL

Address of the MQCLOSE call.

MQCMIT_Call

Type: PMQ_CMIC_CALL

Address of the MQCMIT call.

MQCONN_Call

Type: PMQ_CONN_CALL

Address of the MQCONN call.

MQCONNX_Call

Type: PMQ_CONNX_CALL

Address of the MQCONNX call.

MQCRTMH_Call

Type: PMQ_CRTMH_CALL

Address of the MQCRTMH call.

MQCTL_Call

Type: PMQ_CTL_CALL

Address of the MQCTL call.

MQDISC_Call

Type: PMQ_DISC_CALL

Address of the MQDISC call.

MQDLTMH_Call

Type: PMQ_DLTMH_CALL

Address of the MQDLTMH call.

MQDLTMP_Call

Type: PMQ_DLTMP_CALL

Address of the MQDLTMP call.

MQGET_Call

Type: PMQ_GET_CALL

Address of the MQGET call.

MQINQ_Call

Type: PMQ_INQ_CALL

Address of the MQINQ call.

MQINQMP_Call

Type: PMQ_INQMP_CALL

Address of the MQINQMP call.

MQMHBUF_Call

Type: PMQ_MHBUF_CALL

Address of the MQMHBUF call.

MQOPEN_Call

Type: PMQ_OPEN_CALL

Address of the MQOPEN call.

MQPUT_Call

Type: PMQ_PUT_CALL

Address of the MQPUT call.

MQPUT1_Call

Type: PMQ_PUT1_CALL

Address of the MQPUT1 call.

MQSET_Call

Type: PMQ_SET_CALL

Address of the MQSET call.

MQSETMP_Call

Type: PMQ_SETMP_CALL

Address of the MQSETMP call.

MQSTAT_Call

Type: PMQ_STAT_CALL

Address of the MQSTAT call.

MQSUB_Call

Type: PMQ_SUB_CALL

Address of the MQSUB call.

MQSUBRQ_Call

Type: PMQ_SUBRQ_CALL

Address of the MQSUBRQ call.

MQXCNVC_Call

Type: PMQ_XCNVC_CALL

Address of the MQXCNVC call.

MQXCLWLN_Call

Type: PMQ_XCLWLN_CALL

Address of the MQXCLWLN call.

MQXDX_Call

Type: PMQ_XDX_CALL

Address of the MQXDX call.

MQXEP_Call

Type: PMQ_XEP_CALL

Address of the MQXEP call.

MQZEP_Call

Type: PMQ_ZEP_CALL

Address of the MQZEP call.

C Declaration

```
struct tagMQIEP {
    MQCHAR4      StrucId;          /* Structure identifier */
    MQLONG       Version;          /* Structure version number */
    MQLONG       StrucLength;      /* Structure length */
    MQLONG       Flags;           /* Flags */
    MQPTR        Reserved;        /* Reserved */
    PMQ_BACK_CALL MQBACK_Call;    /* Address of MQBACK */
    PMQ_BEGIN_CALL MQBEGIN_Call;  /* Address of MQBEGIN */
    PMQ_BUFMH_CALL MQBUFMH_Call;  /* Address of MQBUFMH */
    PMQ_CB_CALL  MQCB_Call;       /* Address of MQCB */
    PMQ_CLOSE_CALL MQCLOSE_Call;  /* Address of MQCLOSE */
    PMQ_CMIT_CALL MQCMIT_Call;    /* Address of MQCMIT */
    PMQ_CONN_CALL MQCONN_Call;    /* Address of MQCONN */
    PMQ_CONNX_CALL MQCONNX_Call;  /* Address of MQCONNX */
    PMQ_CRTMH_CALL MQCRTMH_Call;  /* Address of MQCRTMH */
    PMQ_CTL_CALL  MQCTL_Call;     /* Address of MQCTL */
    PMQ_DISC_CALL MQDISC_Call;    /* Address of MQDISC */
    PMQ_DLTMH_CALL MQDLTMH_Call;  /* Address of MQDLTMH */
    PMQ_DLTMP_CALL MQDLTMP_Call;  /* Address of MQDLTMP */
    PMQ_GET_CALL  MQGET_Call;     /* Address of MQGET */
    PMQ_INQ_CALL  MQINQ_Call;     /* Address of MQINQ */
    PMQ_INQMP_CALL MQINQMP_Call;  /* Address of MQINQMP */
    PMQ_MHBUF_CALL MQMHBUF_Call;  /* Address of MQMHBUF */
    PMQ_OPEN_CALL MQOPEN_Call;    /* Address of MQOPEN */
    PMQ_PUT_CALL  MQPUT_Call;     /* Address of MQPUT */
    PMQ_PUT1_CALL MQPUT1_Call;    /* Address of MQPUT1 */
    PMQ_SET_CALL  MQSET_Call;     /* Address of MQSET */
    PMQ_SETMP_CALL MQSETMP_Call;  /* Address of MQSETMP */
    PMQ_STAT_CALL MQSTAT_Call;    /* Address of MQSTAT */
    PMQ_SUB_CALL  MQSUB_Call;     /* Address of MQSUB */
}
```

```

PMQ_SUBRQ_CALL    MQSUBRQ_Call;    /* Address of MQSUBRQ */
PMQ_XCLWLN_CALL   MQXCLWLN_Call;   /* Address of MQXCLWLN */
PMQ_XCNVC_CALL    MQXCNVC_Call;    /* Address of MQXCNVC */
PMQ_XDX_CALL      MQXDX_Call;      /* Address of MQXDX */
PMQ_XEP_CALL      MQXEP_Call;      /* Address of MQXEP */
PMQ_ZEP_CALL      MQZEP_Call;      /* Address of MQZEP */
};

```



Data-conversion exit reference

For z/OS, you must write data-conversion exits in assembler language. For other platforms, it is recommended that you use the C programming language.

To help you to create a data-conversion exit program, the following are supplied:

- A skeleton source file
- A convert characters call
- A utility that creates a fragment of code that performs data conversion on data type structures This utility takes C input only. On z/OS, it produces assembler code.

For the procedure for writing the programs see:

-  [Writing a data-conversion exit program for IBM MQ for IBM i](#)
-  [Writing a data-conversion exit program for IBM MQ for z/OS](#)
- [Writing a data-conversion exit for IBM MQ on UNIX and Linux systems](#)
- [Writing a data-conversion exit for IBM MQ for Windows](#)

Skeleton source file

These can be used as your starting point when writing a data-conversion exit program.

The files supplied are listed in [Table 638 on page 1468](#).

Table 638. Skeleton source files	
Platform	File
AIX	amqsvfc0.c
IBM i	QMQMSAMP/QCSRC(AMQSVFC4)
HP-UX	amqsvfc0.c
Linux	amqsvfc0.c
z/OS	CSQ4BAX8 (“1” on page 1469) CSQ4BAX9 (“2” on page 1469) CSQ4CAX9 (“3” on page 1469)
Solaris	amqsvfc0.c
Windows systems	amqsvfc0.c

Table 638. Skeleton source files (continued)	
Platform	File
Notes: <ol style="list-style-type: none"> 1. Illustrates the MQXCVNC call. 2. A wrapper for the code fragments generated by the utility for use in all environments except CICS. 3. A wrapper for the code fragments generated by the utility for use in the CICS environment. 	

Convert characters call

Use the MQXCNVC (convert characters) call from within a data-conversion exit program to convert character message data from one character set to another. For certain multibyte character sets (for example, UCS-2 character sets), the appropriate options must be used.

No other MQI calls can be made from within the exit; an attempt to make such a call fails with reason code MQRC_CALL_IN_PROGRESS.

See “MQXCNVC - Convert characters” on page 882 for further information on the MQXCNVC call and appropriate options.

Utility for creating conversion-exit code

Use this information to learn more about creating conversion-exit code.

The commands for creating conversion-exit code are:

IBM i

CVTMQMMDTA (Convert IBM MQ Data Type)

Windows, UNIX and Linux systems

crtmqcvx (Create IBM MQ conversion-exit)

z/OS

z/OS

CSQUCVX

The command for your platform produces a fragment of code that performs data conversion on data type structures, for use in your data-conversion exit program. The command takes a file containing one or more C language structure definitions.

z/OS

 On z/OS, it then generates a data set containing assembler code fragments and conversion functions. On other platforms, it generates a file with a C function to convert each structure definition. On z/OS, the utility requires access to the LE/370 runtime library SCEERUN.

Invoking the CSQUCVX utility on z/OS

z/OS

Figure 36 on page 1469 shows an example of the JCL used to invoke the CSQUCVX utility.

```
//CVX      EXEC  PGM=CSQUCVX
//STEPLIB DD  DISP=SHR,DSN=thlqual.SCSQANLE
//          DD  DISP=SHR,DSN=thlqual.SCSQLOAD
//          DD  DISP=SHR,DSN=le370qual.SCEERUN
//SYSPRINT DD  SYSOUT=*
//CSQUINP DD  DISP=SHR,DSN=MY.MQSERIES.FORMATS(MSG1)
//CSQUOUT DD  DISP=OLD,DSN=MY.MQSERIES.EXIT(S(MSG1)
```

Figure 36. Sample JCL used to invoke the CSQUCVX utility

z/OS data definition statements



The CSQUCVX utility requires DD statements with the following DDnames:

Table 639. Data definition statement names and descriptions	
DD statement	Description
SYSPRINT	Specifies a data set or print spool class for reports and error messages.
CSQUINP	Specifies the partitioned data set containing the definitions of the data structures to be converted.
CSQUOUT	Specifies the partitioned data set where the conversion code fragments are to be written. The logical record length (LRECL) must be 80 and the record format (RECFM) must be FB.

Error messages in Windows, UNIX and Linux systems

The `crtmqcvx` command returns messages in the range AMQ7953 through AMQ7970.

These messages are listed in [Messages and reason codes IBM MQ Messages](#).

There are two main types of error:

- Major errors, such as syntax errors, when processing cannot continue.

A message is displayed on the screen giving the line number of the error in the input file. The output file might have been partially created.

- Other errors when a message is displayed stating that a problem has been found but that parsing of the structure can continue.

The output file has been created and contains error information about the problems that have occurred. This error information is prefixed by `#error` so that the code produced is not accepted by any compiler without intervention to rectify the problems.

Valid syntax

Your input file for the utility must conform to the C language syntax.

If you are unfamiliar with C, refer to the [C example](#) in this topic.

In addition, be aware of the following rules:

- `typedef` is recognized only before the `struct` keyword.
- A structure tag is required on your structure declarations.
- You can use empty square brackets `[]` to denote a variable length array or string at the end of a message.
- Multidimensional arrays and arrays of strings are not supported.
- The following additional data types are recognized:
 - `MQBOOL`
 - `MQBYTE`
 - `MQCHAR`
 - `MQFLOAT32`
 - `MQFLOAT64`
 - `MQSHORT`
 - `MQLONG`

- MQINT8
- MQUINT8
- MQINT16
- MQUINT16
- MQINT32
- MQUINT32
- MQINT64
- MQUINT64

MQCHAR fields are code page converted, but MQBYTE, MQINT8 and MQUINT8 are left untouched. If the encoding is different, MQSHORT, MQLONG, MQINT16, MQUINT16, MQINT32, MQUINT32, MQINT64, MQUINT64, MQFLOAT32, MQFLOAT64 and MQBOOL are converted accordingly.

- Do *not* use the following types of data:

- double
- pointers
- bit-fields

This is because the utility for creating conversion-exit code does not provide the facility to convert these data types. To overcome this, you can write your own routines and call them from the exit.

Other points to note:

- Do not use sequence numbers in the input data set.
- If there are fields for which you want to provide your own conversion routines, declare them as MQBYTE, and then replace the generated CMQXCFBA macros with your own conversion code.

C example

```
struct TEST { MQLONG    SERIAL_NUMBER;
              MQCHAR    ID[5];
              MQINT16   VERSION;
              MQBYTE    CODE[4];
              MQLONG    DIMENSIONS[3];
              MQCHAR    NAME[24];
            } ;
```

This corresponds to the following declarations in other programming languages:

COBOL

```
10 TEST.
  15 SERIAL-NUMBER PIC S9(9) BINARY.
  15 ID            PIC X(5).
  15 VERSION      PIC S9(4) BINARY.
  * CODE IS NOT TO BE CONVERTED
  15 CODE         PIC X(4).
  15 DIMENSIONS   PIC S9(9) BINARY OCCURS 3 TIMES.
  15 NAME         PIC X(24).
```

System/390

```
TEST          EQU *
SERIAL_NUMBER DS F
ID            DS CL5
VERSION       DS H
CODE          DS XL4
DIMENSIONS    DS 3F
NAME          DS CL24
```

PL/I

Supported on z/OS only

```
DCL 1 TEST,
  2 SERIAL_NUMBER  FIXED BIN(31),
  2 ID              CHAR(5),
  2 VERSION         FIXED BIN(15),
  2 CODE            CHAR(4),      /* not to be converted */
  2 DIMENSIONS(3)   FIXED BIN(31),
  2 NAME            CHAR(24);
```

MQ_PUBLISH_EXIT - Publish exit

The MQ_PUBLISH_EXIT call can inspect and alter messages delivered to subscribers.

Purpose

Use the publish exit to inspect and alter messages delivered to subscribers:

- Examine the contents of a message published to each subscriber
- Modify the contents of a message published to each subscriber
- Alter the queue to which a message is put
- Stop the delivery of a message to a subscriber

This exit is not available on IBM MQ for z/OS.

Syntax

MQ_PUBLISH_EXIT (*ExitParms*, *PubContext*, *SubContext*)

Parameters

ExitParms (MQPSXP) - Input/Output

ExitParms contains information about the invocation of the exit.

PubContext (MQPBC) - Input

PubContext contains contextual information about the publisher of the publication.

SubContext (MQSBC) - Input/Output

SubContext contains contextual information about the subscriber receiving the publication.

MQPSXP - Publish exit data structure

The MQPSXP structure describes the information that is passed to and returned from the publish exit.

Table 640 on page 1472 summarizes the fields in the structure:

Table 640. Fields in MQPSXP	
Field	Description
<u>StrucID</u>	Structure identifier
<u>Version</u>	Structure version number
<u>ExitId</u>	Type of exit that is being called
<u>ExitReason</u>	Reason for calling the exit
<u>ExitResponse</u>	Response from the exit
<u>ExitResponse2</u>	Secondary response from exit
<u>Feedback</u>	Feedback code

Table 640. Fields in MQPSXP (continued)	
Field	Description
<u>ExitUserArea</u>	Exit user area
<u>ExitData</u>	Exit data
<u>QMgrName</u>	Name of local queue manager
<u>Hconn</u>	Connection handle
<u>MsgDescPtr</u>	Address of message descriptor (MQMD)
<u>MsgHandle</u>	Handle to message properties (MQHMSG)
<u>MsgInPtr</u>	Address of input message
<u>MsgInLength</u>	Length of input message
<u>MsgOutPtr</u>	Address of output message
<u>MsgOutLength</u>	Length of output message
<u>pEntryPoints</u>	Address of the MQIEP structure

Fields

StrucID (MQCHAR4)

StrucID is the structure identifier. The value is as follows:

MQPSXP_STRUCID

MQPSXP_STRUCID is the identifier for the publish exit parameter structure. For the C programming language, the constant MQPSXP_STRUC_ID_ARRAY is also defined; it has the same value as MQPSXP_STRUC_ID, but is an array of characters instead of a string.

StrucID is an input field to the exit.

Version (MQLONG)

Version is the structure version number. The value is as follows:

MQPSXP_VERSION_1

MQPSXP_VERSION_1 is the Version 1 publish exit parameter structure. The constant MQPSXP_CURRENT_VERSION is also defined with the same value.

Version is an input field to the exit.

ExitId (MQLONG)

ExitId is the type of exit that is being called. The value is as follows:

MQXT_PUBLISH_EXIT

Publish exit.

ExitId is an input field to the exit.

ExitReason (MQLONG)

ExitReason is the reason for calling the exit. The possible values are:

MQXR_INIT

The exit for this connection is called for initialization. The exit might acquire and initialize the resources that it needs; for example, main storage.

MQXR_TERM

The exit for this connection is called because the exit is about to be stopped. The exit must free any resources that it has acquired since it was initialized; for example, main storage.

MQXR_PUBLICATION

The exit is called by the queue manager before it puts a publication onto a message queue of a subscriber. The exit can change the message, not put the message on the queue, or halt publication.

ExitReason is an input field to the exit.

ExitResponse (MQLONG)

Set *ExitResponse* in the exit to specify how processing must continue. *ExitResponse* is one of the following values:

MQXCC_OK

Set MQXCC_OK to continue processing normally. Set MQXCC_OK in response to any values of *ExitReason*.

If *ExitReason* has the value MQXR_PUBLICATION, the *DestinationQName* and *DestinationQMGrName* fields of the MQSBC structure identify the destination to which the message is sent.

MQXCC_FAILED

Set MQXCC_FAILED to stop the publish operation. The completion code MQCC_FAILED and reason code 2557 (09FD) (RC2557): MQRC_PUBLISH_EXIT_ERROR is set on return from the exit.

MQXCC_SUPPRESS_FUNCTION

Set MQXCC_SUPPRESS_FUNCTION to stop normal processing of the message. Only set MQXCC_SUPPRESS_FUNCTION if *ExitReason* has the value MQXR_PUBLICATION.

The message continues to be processed by the queue manager according to the MQRO_DISCARD_MSG option in the *Report* field in the message descriptor of the message.

- If the MQRO_DISCARD_MSG option is specified, the message is not delivered to the subscriber.
- If the MQRO_DISCARD_MSG option is not specified, the message is placed on the dead-letter queue. If there is no dead-letter queue, or the message cannot be placed successfully on the dead-letter queue, the publication is not delivered to the subscriber. The delivery of the publication to other subscribers depends on the values of the PMSGDLV and NPMSGDLV topic object attributes. For an explanation of these attributes, see the parameter descriptions for the [DEFINE TOPIC](#) command.

ExitResponse is an output field from the exit.

ExitResponse2 (MQLONG)

ExitResponse2 is reserved for future use.

Feedback (MQLONG)

Feedback is the feedback code to be used if the exit returns MQXCC_SUPPRESS_FUNCTION in *ExitResponse*.

On input to the exit, *Feedback* always has the value MQFB_NONE. If the exit returns MQXCC_SUPPRESS_FUNCTION, set *Feedback* to the value to be used for the message when the queue manager places it on the dead-letter queue. On return from the exit, if *Feedback* has the original value MQFB_NONE, the queue manager sets *Feedback* to MQFB_STOPPED_BY_PUBSUB_EXIT.

Feedback is an input/output field to the exit.

ExitUserArea (MQBYTE16)

ExitUserArea is a field that is available for the exit to use. Each connection has a separate *ExitUserArea*. The length of *ExitUserArea* is given by MQ_EXIT_USER_AREA_LENGTH.

The *ExitReason* field has the value MQXR_INIT on the first invocation of the exit. *ExitUserArea* is initialized to MQXUA_NONE on the first invocation of the exit for a connection. Subsequent changes to *ExitUserArea* are preserved across invocations of the exit.

ExitUserArea is an input/output field to the exit.

ExitData (MQCHAR32)

ExitData is fixed exit data defined by the *PublishExitData* parameter of the stanza in the initialization file of the queue manager. The data is padded with blanks to the full length of the field. If there is no fixed exit data defined in the initialization file, *ExitData* is blank. The length of *ExitData* is given by MQ_EXIT_DATA_LENGTH.

ExitData is an input field to the exit.

QMgrName (MQCHAR48)

QMgrName is the name of the local queue manager. The name is padded with blanks to the full length of the field. The length of this field is given by MQ_Q_MGR_NAME_LENGTH.

QMgrName is an input field to the exit.

Hconn (MQHCONN)

Hconn is the handle representing a connection to the queue manager. Only use *Hconn* as a parameter to the MQSETMP, MQINQMMP, or MQDLTMP message property function calls to work with message properties.

Hconn is an input field to the exit.

MsgDescPtr (PMQMD)

MsgDescPtr is the address of message descriptor (MQMD) of the message being processed, and is a copy of the MQMD returned from the MQPUT call. The exit can change the contents of the message descriptor. Any change to the contents of the message descriptor must be done with care. In particular, in the case where the *SubType* field of the MQSBC structure is of value MQSUBTYPE_PROXY, the *CorrelId* field in the message descriptor must not be changed.

No message descriptor is passed to the exit if *ExitReason* is MQXR_INIT or MQXR_TERM; in these cases, *MsgDescPtr* is the null pointer.

MsgDescPtr is an input field to the exit.

MsgHandle (MQHMSG)

MsgHandle is the handle to message properties. Only use *MsgHandle* with the MQSETMP, MQINQMMP, or MQDLTMP message property function calls to work with message properties.

MsgHandle is an input field to the exit.

MsgInPtr (PMQVOID)

MsgInPtr is the address of the input message data. The contents of the buffer addressed by *MsgInPtr* can be modified by the exit; see [MsgOutPtr](#).

MsgInPtr is an input field to the exit.

MsgInLength (MQLONG)

MsgInLength is the length in bytes of the message data passed to the exit. The address of the data is given by *MsgInPtr*.

MsgInLength is an input field to the exit.

MsgOutPtr (PMQVOID)

MsgOutPtr is the address of a buffer containing message data that is returned from the exit. On entry to the exit, *MsgOutPtr* is null. On return from the exit, if the value is still null, the queue manager sends the message specified by *MsgInPtr*, with the length given by *MsgInLength*.

If the exit modifies the message data, use one of the following procedures:

- If the length of the data does not change, the data can be modified in the buffer addressed by *MsgInPtr*. In this case, do not change *MsgOutPtr* and *MsgOutLength*.
- If the modified data is shorter than the original data, the data can be modified in the buffer addressed by *MsgInPtr*. In this case *MsgOutPtr* must be set to the address of the input message buffer, and *MsgOutLength* set to the new length of the message data.
- If the modified data is, or might be, longer than the original data, the exit must obtain a new message buffer. Copy the modified data into it. Set *MsgOutPtr* to the address of the new buffer, and

set *MsgOutLength* to the length of the new message data. The exit is responsible for freeing the buffer addressed by *MsgOutPtr* when the exit is next called.

Note: *MsgOutPtr* is always the null pointer on input to the exit, and not the address of a previously obtained message buffer. To free the previously obtained buffer, the exit must save its address and length. Save the information either in *ExitUserArea*, or in a control block that has its address saved in *ExitUserArea*.

MsgOutPtr is an input/output field to the exit.

MsgOutLength (MQLONG)

MsgOutLength is the length in bytes of the message data returned by the exit. On input to the exit, this field is always zero. On return from the exit, this field is ignored if *MsgOutPtr* is null. See *MsgOutPtr* for information about modifying the message data.

MsgOutLength is an input/output field to the exit.

pEntryPoints (PMQIEP)

pEntryPoints is the address of an MQIEP structure through which MQI and DCI calls can be made.

C language declaration - MQPSXP

```
typedef struct tagMQPSXP {
    MQCHAR4      StrucId;           /* Structure identifier */
    MQLONG       Version;          /* Structure version number */
    MQLONG       ExitId;           /* Type of exit */
    MQLONG       ExitReason;        /* Reason for invoking exit */
    MQLONG       ExitResponse;      /* Response from exit */
    MQLONG       ExitResponse2;     /* Reserved */
    MQLONG       Feedback;          /* Feedback code */
    MQBYTE16     ExitUserArea;      /* Exit user area */
    MQCHAR32     ExitData;          /* Exit data */
    MQCHAR48     QMgrName;          /* Name of local queue manager */
    MQHCONN      Hconn;            /* Connection handle */
    MQHMSG       MsgHandle;         /* Handle to message properties */
    PMQMD        MsgDescPtr;        /* Address of message descriptor */
    PMQVOID      MsgInPtr;          /* Address of input message data */
    MQLONG       MsgInLength;       /* Length of input message data */
    PMQVOID      MsgOutPtr;         /* Address of output message data */
    MQLONG       MsgOutLength;      /* Length of output message data */
    /* Ver:1 */
    PMQIEP       pEntryPoints;      /* Address of the MQIEP structure */
    /* Ver:2 */
} MQPSXP;
```

MQPBC - Publication context data structure

The MQPBC structure contains the contextual information, relating to the publisher of the publication, that is passed to the publish exit.

Table 641 on page 1476 summarizes the fields in the structure:

Table 641. Fields in MQPBC	
Field	Description
<u>StrucID</u>	Structure identifier
<u>Version</u>	Structure version number
<u>PubTopicString</u>	Publish topic string
<u>MsgDescPtr</u>	Address of message descriptor (MQMD)

Fields

StrucID (MQCHAR4)

StrucID is the structure identifier. The value is as follows:

MQPBC_STRUCID

MQPBC_STRUCID is the identifier for the publication context structure. For the C programming language, the constant MQPBC_STRUC_ID_ARRAY is also defined; it has the same value as MQPBC_STRUC_ID, but is an array of characters instead of a string.

StrucID is an input field to the exit.

Version (MQLONG)

Version is the structure version number. The value is as follows:

MQPBC_VERSION_1

MQPBC_VERSION_1 is the Version 1 publish exit parameter structure.

MQPBC_VERSION_2

MQPBC_VERSION_2 is the Version 2 publish exit parameter structure. The constant MQPBC_CURRENT_VERSION is also defined with the same value.

Version is an input field to the exit.

PubTopicString (MQCHARV)

PubTopicString is the topic string being published to.

PubTopicString is an input field to the exit.

MsgDescPtr (PMQMD)

MsgDescPtr is the address of a copy of the message descriptor (MQMD) for the message being processed.

MsgDescPtr is an input field to the exit.

C language declaration - MQPBC

```
typedef struct tagMQPBC {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQCHARV    PubTopicString;    /* Publish topic string */
    PMQMD      MsgDescPtr;       /* Address of message descriptor */
} MQPBC;
```

MQSBC - Subscription context data structure

The MQSBC structure contains the contextual information, relating to the subscriber that is receiving the publication, that is passed to the publish exit.

Table 642 on page 1477 summarizes the fields in the structure:

Table 642. Fields in MQSBC	
Field	Description
<i>StrucID</i>	Structure identifier
<i>Version</i>	Structure version number
<i>DestinationQMgrName</i>	Name of destination queue manager
<i>DestinationQName</i>	Name of destination queue
<i>SubType</i>	Type of subscription
<i>SubOptions</i>	Subscription options
<i>ObjectName</i>	Object name
<i>ObjectString</i>	Object string
<i>SubTopicString</i>	Subscription topic string
<i>SubName</i>	Subscription name

Table 642. Fields in MQSBC (continued)	
Field	Description
<i>SubId</i>	Subscription identifier
<i>SelectionString</i>	Address of selection string
<i>SubLevel</i>	Subscription level
<i>PSPProperties</i>	Publish/subscribe properties

Fields

StrucID (MQCHAR4)

Structure identifier. The value is as follows:

MQSBC_STRUCID

MQSBC_STRUCID is the identifier for the publish exit parameter structure. For the C programming language, the constant MQSBC_STRUC_ID_ARRAY is also defined; MQSBC_STRUC_ID_ARRAY has the same value as MQSBC_STRUC_ID, but is an array of characters instead of a string.

StrucID is an input field to the exit.

Version (MQLONG)

Structure version number. The value is as follows:

MQSBC_VERSION_1

Version 1 publish exit parameter structure. The constant MQSBC_CURRENT_VERSION is also defined with the same value.

Version is an input field to the exit.

DestinationQMGrName (MQCHAR48)

DestinationQMGrName is the name of the queue manager to which the message is being sent. The name is padded with blanks to the full length of the field. The name can be altered by the exit. The length of this field is given by MQ_Q_MGR_NAME_LENGTH.

DestinationQMGrName is an input/output field to the exit; see [note](#).

DestinationQName (MQCHAR48)

DestinationQName is the name of the queue to which the message is being sent. The name is padded with blanks to the full length of the field. The name can be altered by the exit. The length of this field is given by MQ_Q_NAME_LENGTH.

DestinationQName is an input/output field to the exit; see [note](#).

SubType (MQLONG)

SubType indicates how the subscription was created. Valid values are MQSUBTYPE_API, MQSUBTYPE_ADMIN and MQSUBTYPE_PROXY ; see [Inquire Subscription Status \(Response\)](#).

SubType is an input field to the exit.

SubOptions (MQLONG)

SubOptions are the subscription options; see “Options (MQLONG)” on [page 542](#) for a description of values this field can take.

SubOptions is an input field to the exit.

ObjectName (MQCHAR48)

ObjectName is the name of the topic object as defined on the local queue manager. The length of this field is given by MQ_TOPIC_NAME_LENGTH. The object name is the name of the administrative topic object that the queue manager has associated with the topic string. Even if the subscriber provided a topic object as part of the subscription, the *ObjectName* might be a different topic object. The association of a topic object with a subscription depends upon the full resolution of *SubTopicString*.

ObjectName is an input field to the exit.

ObjectString (MQCHARV)

ObjectString is the full topic string of the publication that was subscribed to. Any wildcards in the original subscription string are resolved. It is different to the MQSD subscription *ObjectString* field described in [“ObjectString \(MQCHARV\)”](#) on page 541, which might contain wildcards, and is exclusive of any object name provided by the subscriber.

ObjectString is an input field to the exit.

SubTopicString (MQCHARV)

SubTopicString is the complete topic string as supplied by the subscriber. *SubTopicString* is the combination of the topic string defined in a topic object, and a topic string. A subscriber must provide either a topic object, a topic string, or both. If the subscriber provides a topic string, it might contain wildcards.

SubTopicString is an input field to the exit.

SubName (MQCHARV)

SubName is the subscription name that is provided either by the subscriber, or is a generated name.

SubName is an input field to the exit.

SubId (MQBYTE 24)

SubId is the unique internal subscription identifier.

SubId is an input field to the exit.

SelectionString (MQCHARV)

SelectionString is the selection criteria used when subscribing for messages from a topic; see [Selectors](#).

SelectionString is an input field to the exit.

SubLevel (MQLONG)

SubLevel is the interception level associated with the subscription; see [“SubLevel \(MQLONG\)”](#) on page 552 for further details.

SubLevel is an input field to the exit.

PSPProperties (MQLONG)

PSPProperties are the publish/subscribe properties. They specify how publish/subscribe related message properties are added to messages sent to this subscription. Possible values are MQPSPROP_NONE, MQPSPROP_COMPAT, MQPSPROP_RFH2, MQPSPROP_MSGPROP. See [Optional parameters \(Change, Copy, and Create Subscription\)](#) for a description of these values.

PSPProperties is an input field to the exit.

Note: Authorization checks are only performed on the original values of *DestinationQMGrName* and *DestinationQName* before they are passed to the publish exit. No new authorization checks are performed when the exit changes the destination queue, either by changing *DestinationQMGrName* or *DestinationQName*.

C language declaration - MQSBC

```
typedef struct tagMQSBC {
    MQCHAR4      StructId;           /* Structure identifier */
    MQLONG       Version;            /* Structure version number */
    MQCHAR48     DestinationQMGrName; /* Destination queue manager */
    MQCHAR48     DestinationQName;   /* Destination queue name */
    MQLONG       SubType;            /* Type of subscription */
    MQLONG       SubOptions;         /* Subscription options */
    MQCHAR48     ObjectName;         /* Object name */
    MQCHARV      ObjectString;       /* Object string */
    MQCHARV      SubTopicString;     /* Subscription topic string */
    MQCHARV      SubName;           /* Subscription name */
    MQBYTE24     SubId;             /* Subscription identifier */
    MQCHARV      SelectionString;    /* Subscription selection string */
    MQLONG       SubLevel;          /* Subscription level */
}
```

```

MQLONG      PSProperties;      /* Publish/subscribe properties */
} MQSBC;

```

Channel-exit calls and data structures

This collection of topics provide reference information about the special IBM MQ calls and data structures that you can use when you write channel exit programs.

This information is product-sensitive programming interface information. You can write IBM MQ user exits in the following programming languages:

Platform	Programming languages
IBM MQ for z/OS	Assembler and C (which must conform to the C system programming environment for system exits, described in the <i>z/OS C/C++ Programming Guide</i> .)
IBM MQ for IBM i	ILE C, ILE COBOL, and ILE RPG
All other IBM MQ platforms	C

You can also write user exits in Java for use only with Java and JMS applications. For more information about creating and using channel exits with the IBM MQ classes for Java, see [Using channel exits in IBM MQ classes for Java](#) and for IBM MQ classes for JMS, see [Using channel exits with IBM MQ classes for JMS](#).

You cannot write IBM MQ user exits in TAL or Visual Basic. However, a declaration for the MQCD structure is provided in Visual Basic for use on the MQCONN call from an IBM MQ MQI client program.

In a number of cases in the descriptions that follow, parameters are arrays or character strings with a size that is not fixed. For these parameters, a lowercase "n" is used to represent a numeric constant. When the declaration for that parameter is coded, the "n" must be replaced by the numeric value required. For further information about the conventions used in these descriptions, see the [“Elementary data types”](#) on [page 231](#).

Data definition files

Data definition files are supplied with IBM MQ for each of the supported programming languages. For details of these files, see [Copy, header, include, and module files](#).

MQ_CHANNEL_EXIT - Channel exit

The MQ_CHANNEL_EXIT call describes the parameters that are passed to each of the channel exits called by the Message Channel Agent.

No entry point called MQ_CHANNEL_EXIT is provided by the queue manager; the name MQ_CHANNEL_EXIT is of no special significance since the names of the channel exits are provided in the channel definition MQCD.

There are five types of channel exit:

- Channel security exit
- Channel message exit
- Channel send exit
- Channel receive exit
- Channel message-retry exit

The parameters are similar for each type of exit, and the description given here applies to all of them, except where specifically noted.

Syntax

MQ_CHANNEL_EXIT (*ChannelExitParms*, *ChannelDefinition*, *DataLength*,
AgentBufferLength, *AgentBuffer*, *ExitBufferLength*, *ExitBufferAddr*)

Parameters

The MQ_CHANNEL_EXIT call has the following parameters.

ChannelExitParms (MQCXP) - input/output

Channel exit parameter block.

This structure contains additional information relating to the invocation of the exit. The exit sets information in this structure to indicate how the MCA proceeds.

ChannelDefinition (MQCD) - input/output

Channel definition.

This structure contains parameters set by the administrator to control the behavior of the channel.

DataLength (MQLONG) - input/output

Length of data.

The data depends on the type of exit:

- For a channel security exit, when the exit is invoked this parameter contains the length of any security message in the *AgentBuffer* field, if *ExitReason* is MQXR_SEC_MSG. It is zero if there is no message. The exit must set this field to the length of any security message to be sent to its partner if it sets *ExitResponse* to MQXCC_SEND_SEC_MSG or MQXCC_SEND_AND_REQUEST_SEC_MSG. The message data is in either *AgentBuffer* or *ExitBufferAddr*.

The content of security messages is the sole responsibility of the security exits.
- For a channel message exit, when the exit is invoked this parameter contains the length of the message (including the transmission queue header). The exit must set this field to the length of the message in either *AgentBuffer* or *ExitBufferAddr* that is to proceed. This must be greater than or equal to the length of the transmission queue header (MQXQH).
- For a channel send or channel receive exit, when the exit is invoked this parameter contains the length of the transmission. The exit must set this field to the length of the transmission in either *AgentBuffer* or *ExitBufferAddr* that is to proceed.

If a security exit sends a message, and there is no security exit at the other end of the channel, or the other end sets an *ExitResponse* of MQXCC_OK, the initiating exit is re-invoked with MQXR_SEC_MSG and a null response (*DataLength*=0).

AgentBufferLength (MQLONG) - input

Length of agent buffer.

This parameter can be greater than *DataLength* on invocation.

For channel message, send, and receive exits, any unused space on invocation can be used by the exit to expand the data in place. If this is done, the *DataLength* parameter must be set appropriately by the exit.

In the C programming language, this parameter is passed by address.

AgentBuffer (MQBYTE x AgentBufferLength) - input/output

Agent buffer.

The contents of this parameter depend upon the exit type:

- For a channel security exit, on invocation of the exit it contains a security message if *ExitReason* is MQXR_SEC_MSG. To send a security message back, the exit can either use this buffer or its own buffer (*ExitBufferAddr*).
- For a channel message exit, on invocation of the exit this parameter contains:
 - The transmission queue header (MQXQH), which includes the message descriptor (which itself contains the context information for the message), immediately followed by
 - The message data

If the message is to proceed, the exit can do one of the following:

- Leave the contents of the buffer untouched
- Modify the contents in place (returning the new length of the data in *DataLength* ; this must not be greater than *AgentBufferLength*)
- Copy the contents to the *ExitBufferAddr*, making any required changes

Any changes that the exit makes to the transmission queue header are not checked; however, erroneous modifications might mean that the message cannot be put at the destination.

- For a channel send or receive exit, on invocation of the exit this contains the transmission data. The exit can do one of the following:
 - Leave the contents of the buffer untouched
 - Modify the contents in place (returning the new length of the data in *DataLength* ; this must not be greater than *AgentBufferLength*)
 - Copy the contents to the *ExitBufferAddr*, making any required changes

The first 8 bytes of the data must not be changed by the exit.

ExitBufferLength (MQLONG) - input/output

Length of exit buffer.

On the first invocation of the exit, this parameter is set to zero. Thereafter whatever value is passed back by the exit, on each invocation, is presented to the exit next time it is invoked. The value is not used by the MCA.

Note: This parameter must not be used by exits written in programming languages which do not support the pointer data type.

ExitBufferAddr (MQPTR) - input/output

Address of exit buffer.

This parameter is a pointer to the address of a buffer of storage managed by the exit, where it can choose to return message or transmission data (depending upon the type of exit) to the agent if the buffer of the agent is or might not be large enough, or if it is more convenient for the exit to do so.

On the first invocation of the exit, the address passed to the exit is null. Thereafter whatever address is passed back by the exit, on each invocation, is presented to the exit the next time it is invoked.

If *ExitBufferAddr* is null the data used is taken from the *AgentBuffer* parameter.

If *ExitBufferAddr* is not null the data used is taken from the buffer pointed to by the *ExitBufferAddr* parameter.

Note: This parameter must not be used by exits written in programming languages that do not support the pointer data type.

C invocation

```
exitname (&ChannelExitParms, &ChannelDefinition,
&DataLength, &AgentBufferLength, AgentBuffer,
&ExitBufferLength, &ExitBufferAddr);
```

The parameters passed to the exit are declared as follows:

```
MQCXP  ChannelExitParms;  /* Channel exit parameter block */
MQCD   ChannelDefinition; /* Channel definition */
MQLONG DataLength;        /* Length of data */
MQLONG AgentBufferLength; /* Length of agent buffer */
MQBYTE AgentBuffer[n];    /* Agent buffer */
MQLONG ExitBufferLength;  /* Length of exit buffer */
MQPTR  ExitBufferAddr;    /* Address of exit buffer */
```

COBOL invocation

```
CALL 'exitname' USING CHANNELEXITPARMS, CHANNELDEFINITION,
                     DATALENGTH, AGENTBUFFERLENGTH, AGENTBUFFER,
                     EXITBUFFERLENGTH, EXITBUFFERADDR.
```

The parameters passed to the exit are declared as follows:

```
** Channel exit parameter block
01 CHANNELEXITPARMS.
   COPY CMQCXPV.
** Channel definition
01 CHANNELDEFINITION.
   COPY CMQCDV.
** Length of data
01 DATALENGTH      PIC S9(9) BINARY.
** Length of agent buffer
01 AGENTBUFFERLENGTH PIC S9(9) BINARY.
** Agent buffer
01 AGENTBUFFER       PIC X(n).
** Length of exit buffer
01 EXITBUFFERLENGTH  PIC S9(9) BINARY.
** Address of exit buffer
01 EXITBUFFERADDR    POINTER.
```

RPG invocation (ILE)

```
C*..1.....2.....3.....4.....5.....6.....7..
C                                CALLP      exitname(MQCXP : MQCD : DATLEN :
C                                ABUFL : ABUF : EBUFL :
C                                EBUF)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
Dexitname      PR                EXTPROC('exitname')
D* Channel exit parameter block
D MQCXP                160A
D* Channel definition
D MQCD                1328A
D* Length of data
D DATLEN              10I 0
D* Length of agent buffer
D ABUFL              10I 0
D* Agent buffer
D ABUF                *   VALUE
D* Length of exit buffer
D EBUFL              10I 0
D* Address of exit buffer
D EBUF                *
```

System/390 assembler invocation

```
CALL EXITNAME, (CHANNELEXITPARMS, CHANNELDEFINITION, DATALENGTH, X
               AGENTBUFFERLENGTH, AGENTBUFFER, EXITBUFFERLENGTH, X
               EXITBUFFERADDR)
```

The parameters passed to the exit are declared as follows:

CHANNELEXITPARMS	CMQCXPA	,	Channel exit parameter block
CHANNELDEFINITION	CMQCDA	,	Channel definition
DATALength	DS	F	Length of data
AGENTBUFFERLENGTH	DS	F	Length of agent buffer
AGENTBUFFER	DS	CL(n)	Agent buffer
EXITBUFFERLENGTH	DS	F	Length of exit buffer
EXITBUFFERADDR	DS	F	Address of exit buffer

Usage notes

1. The function performed by the channel exit is defined by the provider of the exit. The exit, however, must conform to the rules defined here and in the associated control block, the MQCXP.
2. The *ChannelDefinition* parameter passed to the channel exit might be one of several versions. See the *Version* field in the MQCD structure for more information.
3. If the channel exit receives an MQCD structure with the *Version* field set to a value greater than MQCD_VERSION_1, the exit must use the *ConnectionName* field in MQCD, in preference to the *ShortConnectionName* field.
4. In general, channel exits are allowed to change the length of message data. This can arise as a result of the exit adding data to the message, or removing data from the message, or compressing or encrypting the message. However, special restrictions apply if the message is a segment that contains only part of a logical message. In particular, there must be no net change in the length of the message as a result of the actions of complementary sending and receiving exits.

For example, it is permissible for a sending exit to shorten the message by compressing it, but the complementary receiving exit must restore the original length of the message by decompressing it, so that there is no net change in the length of the message.

This restriction arises because changing the length of a segment would cause the offsets of later segments in the message to be incorrect, and this would inhibit the ability of the queue manager to recognize that the segments formed a complete logical message.

MQ_CHANNEL_AUTO_DEF_EXIT - Channel auto-definition exit

The MQ_CHANNEL_AUTO_DEF_EXIT call describes the parameters that are passed to the channel auto-definition exit called by the Message Channel Agent.

No entry point called MQ_CHANNEL_AUTO_DEF_EXIT is provided by the queue manager; the name MQ_CHANNEL_AUTO_DEF_EXIT is of no special significance because the names of the auto-definition exits are provided in the queue manager.

Syntax

MQ_CHANNEL_AUTO_DEF_EXIT (*ChannelExitParms*, *ChannelDefinition*)

Parameters

The MQ_CHANNEL_AUTO_DEF_EXIT call has the following parameters.

ChannelExitParms (MQCXP) - input/output

Channel exit parameter block.

This structure contains additional information relating to the invocation of the exit. The exit sets information in this structure to indicate how the MCA proceeds.

ChannelDefinition (MQCD) - input/output

Channel definition.

This structure contains parameters set by the administrator to control the behavior of channels which are created automatically. The exit sets information in this structure to modify the default behavior set by the administrator.

The MQCD fields listed must not be altered by the exit:

- *ChannelName*
- *ChannelType*
- *StrucLength*
- *Version*

If other fields are changed, the value set by the exit must be valid. If the value is not valid, an error message is written to the error log file or displayed on the console (as appropriate to the environment).



Attention: Auto-defined channels created by a channel automatic definition (CHAD) exit cannot set the certificate label, because the SSL or TLS handshake has occurred by the time the channel is created. Setting the certificate label in a CHAD exit for inbound channels has no effect.

C invocation

```
exitname (&ChannelExitParms, &ChannelDefinition);
```

The parameters passed to the exit are declared as follows:

```
MQCXP  ChannelExitParms;  /* Channel exit parameter block */
MQCD   ChannelDefinition; /* Channel definition */
```

COBOL invocation

```
CALL 'exitname' USING CHANNELEXITPARMS, CHANNELDEFINITION.
```

The parameters passed to the exit are declared as follows:

```
** Channel exit parameter block
01 CHANNELEXITPARMS.
   COPY CMQCXPV.
** Channel definition
01 CHANNELDEFINITION.
   COPY CMQCDV.
```

RPG invocation (ILE)

```
C*..1.....2.....3.....4.....5.....6.....7..
C                                CALLP      exitname(MQCXP : MQCD)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
Dexitname          PR          EXTPROC('exitname')
D* Channel exit parameter block
D MQCXP                                160A
D* Channel definition
D MQCD                                1328A
```

System/390 assembler invocation

```
CALL EXITNAME, (CHANNELEXITPARMS, CHANNELDEFINITION)
```

The parameters passed to the exit are declared as follows:

```
CHANNELEXITPARMS  CMQCXPA  ,  Channel exit parameter block
CHANNELDEFINITION CMQCDA   ,  Channel definition
```


Usage notes

1. The function performed by the channel exit is defined by the provider of the exit. The exit, however, must conform to the rules defined here and in the associated control block, the MQCXP.
2. The *ChannelExitParms* parameter passed to the channel auto-definition exit is an MQCXP structure. The version of MQCXP passed depends on the environment in which the exit is running; see the description of the *Version* field in “MQCXP - Channel exit parameter” on page 1526 for details.
3. The *ChannelDefinition* parameter passed to the channel auto-definition exit is an MQCD structure. The version of MQCD passed depends on the environment in which the exit is running; see the description of the *Version* field in “MQCD - Channel definition” on page 1487 for details.

MQXWAIT - Wait in exit

The MQXWAIT call waits for an event to occur. It can be used only from a channel exit on z/OS.

The use of MQXWAIT helps to avoid performance problems that might otherwise occur if a channel exit does something that causes a wait. The event MQXWAIT is waiting on is signaled by an MVS ECB (event control block). The ECB is described in the MQXWD control block description.

 For more information about the use of MQXWAIT and writing channel-exit programs, see [Writing channel exit programs on z/OS](#)

Syntax

MQXWAIT (*Hconn*, *WaitDesc*, *CompCode*, *Reason*)

Parameters

The MQXWAIT call has the following parameters.

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN call issued in the same or earlier invocation of the exit.

WaitDesc (MQXWD) - input/output

Wait descriptor.

This parameter describes the event to wait for. See “MQXWD - Exit wait descriptor” on page 1540 for details of the fields in this structure.

CompCode (MQLONG) - output

Completion code.

It is one of the following codes:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

MQRC_ADAPTER_NOT_AVAILABLE

(2204, X'89C') Adapter not available.

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent.

MQRC_XWAIT_CANCELED

(2107, X'83B') MQXWAIT call canceled.

MQRC_XWAIT_ERROR

(2108, X'83C') Invocation of MQXWAIT call not valid.

C invocation

```
MQXWAIT (Hconn, &WaitDesc, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;      /* Connection handle */
MQXWD    WaitDesc;   /* Wait descriptor */
MQLONG   CompCode;   /* Completion code */
MQLONG   Reason;     /* Reason code qualifying CompCode */
```

System/390 assembler invocation

```
CALL MQXWAIT,(HCONN,WAITDESC,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN      DS      F      Connection handle
WAITDESC   CMQXWDA  ,      Wait descriptor
COMPCODE    DS      F      Completion code
REASON      DS      F      Reason code qualifying COMPCODE
```

MQCD - Channel definition

The MQCD structure contains the parameters which control execution of a channel. It is passed to each channel exit that is called from a Message Channel Agent (MCA).

For more information about channel exits, see [“MQ_CHANNEL_EXIT - Channel exit” on page 1480](#). The description in this topic relates both to message channels and to MQI channels.

Exit name fields

When an exit is called, the relevant field from *SecurityExit*, *MsgExit*, *SendExit*, *ReceiveExit*, and *MsgRetryExit* contains the name of the exit currently being invoked. The meaning of the name in these fields depends on the environment in which the MCA is running. Except where noted, the name is left-aligned within the field, with no embedded blanks; the name is padded with blanks to the length of the field. In the descriptions that follow, square brackets ([]) denote optional information:

UNIX systems

The exit name is the name of a dynamically loadable module or library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path:

```
[ path ] library ( function )
```

The name is limited to a maximum of 128 characters.

z/OS

The exit name is the name of a load module that is valid for specification on the EP parameter of the LINK or LOAD macro. The name is limited to a maximum of eight characters.

Windows

The exit name is the name of a dynamic-link library, suffixed with the name of a function residing in that library. The function name must be enclosed in parentheses. The library name can optionally be prefixed with a directory path and drive:

```
[d:][ path ] library ( function )
```

The name is limited to a maximum of 128 characters.

IBM i

The exit name is a 10 byte program name followed by a 10 byte library name. If the names are less than 10 bytes long, each name is padded with blanks to make it 10 bytes. The library name can be *LIBL except when calling a channel auto-definition exit, in which case a fully qualified name is required.

Changing MQCD fields in a channel exit

A channel exit can change fields in the MQCD. The changed value remains in the MQCD and is passed to any remaining exits in an exit chain and to any conversation sharing the channel instance. The changed MQCD is also used by the MCA for its normal processing during the continuing lifetime of the channel.

The following MQCD fields must not be altered by the exit:

- ChannelName
- ChannelType
- StrucLength
- Version

Related reference

[“Fields” on page 1488](#)

This topic lists all the fields in the MQCD structure and describes each field.

[“C declaration” on page 1513](#)

This declaration is the C declaration for the MQCD structure.

[“COBOL declaration” on page 1515](#)

This declaration is the COBOL declaration for the MQCD structure.

[“RPG declaration \(ILE\)” on page 1517](#)

This declaration is the RPG declaration for the MQCD structure.

[“System/390 assembler declaration” on page 1520](#)

This declaration is the System/390 assembler declaration for the MQCD structure.

[“Visual Basic declaration” on page 1522](#)

This declaration is the Visual Basic declaration of the MQCD structure.

[“Changing MQCD fields in a channel exit” on page 1523](#)

A channel exit can change fields in the MQCD. However, these changes are not typically acted on, except in the circumstances listed.

Fields

This topic lists all the fields in the MQCD structure and describes each field.

BatchDataLimit (MQLONG)

This field specifies the limit, in kilobytes, of the amount of data that can be sent through a channel before taking a sync point.

A sync point is taken after the message that caused the limit to be reached has flowed across the channel.

The batch is terminated when one of the following conditions is met:

- **BatchSize** messages have been sent.

- **BatchDataLimit** bytes have been sent.
- The transmission queue is empty and **BatchInterval** is exceeded.

The value must be in the range 0 - 999999. The default value is 5000.

A value of zero in this attribute means that no data limit is applied to batches over this channel.

This parameter only applies to channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSRCVR, or MQCHT_CLUSSDR.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_11.

BatchHeartbeat (MQLONG)

This field specifies the time interval that is used to trigger a batch heartbeat for the channel.

Batch heartbeating allows sender channels to determine whether the remote channel instance is still active before going indoubt. A batch heartbeat occurs if a sender channel has not communicated with the remote channel instance within the specified time interval.

The value is in the range 0 through 999 999; the units are milliseconds. A value of zero indicates that batch heartbeating is not enabled.

This field is relevant only for channels that have a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_7.

BatchInterval (MQLONG)

This field specifies the approximate time in milliseconds that a channel keeps a batch open, if fewer than *BatchSize* messages have been transmitted in the current batch.

If *BatchInterval* is greater than zero, the batch is terminated by whichever of the following events occur first:

- *BatchSize* messages have been sent, or
- *BatchInterval* milliseconds have elapsed since the start of the batch.

If *BatchInterval* is zero, the batch is terminated by whichever of the following events occur first:

- *BatchSize* messages have been sent, or
- the transmission queue becomes empty.

BatchInterval must be in the range zero through 999 999 999.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

This is an input field to the exit. The field is not present when *Version* is less than MQCD_VERSION_4.

BatchSize (MQLONG)

This field specifies the maximum number of messages that can be sent through a channel before synchronizing the channel.

This field is not relevant for channels with a *ChannelType* of MQCHT_SVRCONN or MQCHT_CLNTCONN.

CertificateLabel (MQCHAR64)

This field gives details of the certificate label being used.

IBM MQ initializes the default value for the *CertificateLabel* field as blanks.

This is interpreted at runtime as the default value, and is backwards compatible.

For example, specifying a MQCD version less than 11, or using the default value of blanks for the *CertificateLabel* field, means that this field is ignored.

The length of this field is given by MQ_CERT_LABEL_LENGTH.

ChannelMonitoring (MQLONG)

This field specifies the current level of monitoring data collection for the channel.

This field is not relevant for channels with a ChannelType of MQCHT_CLNTCONN.

It is one of the following values:

- MQMON_OFF
- MQMON_LOW
- MQMON_MEDIUM
- MQMON_HIGH

This is an input field to the exit. It is not present if *Version* is less than MQCD_VERSION_8.

ChannelName (MQCHAR20)

This field specifies the channel definition name.

There must be a channel definition of the same name at the remote machine to be able to communicate.

The name must use only the characters:

- Uppercase A-Z
- Lowercase a-z
- Numerics 0-9
- Period (.)
- Forward slash (/)
- Underscore (_)
- Percent sign (%)

and be padded to the right with blanks. Leading or embedded blanks are not allowed.

The length of this field is given by MQ_CHANNEL_NAME_LENGTH.

ChannelStatistics (MQLONG)

This field specifies the current level of statistics data collection for the channel.

This field is not relevant for channels with a ChannelType of MQCHT_CLNTCONN or MQCHT_SVRCONN.

It is one of the following values:

- MQMON_OFF
- MQMON_LOW
- MQMON_MEDIUM
- MQMON_HIGH

This is an input field to the exit. It is not present if *Version* is less than MQCD_VERSION_8.

ChannelType (MQLONG)

This field specifies the type of channel.

It is one of the following values:

MQCHT_SENDER

Sender.

MQCHT_SERVER

Server.

MQCHT_RECEIVER

Receiver.

MQCHT_REQUESTER

Requester.

MQCHT_CLNTCONN

Client connection.

MQCHT_SVRCONN

Server-connection (for use by clients).

MQCHT_CLUSSDR

Cluster sender.

MQCHT_CLUSRCVR

Cluster receiver.

ClientChannelWeight (MQLONG)

This field specifies a weighting to influence which client-connection channel definition is used.

The ClientChannelWeight attribute is used so that client channel definitions can be selected at random based on their weighting when more than one suitable definition is available. When a client issues an MQCONN requesting connection to a queue manager group, by specifying a queue manager name starting with an asterisk, and more than one suitable channel definition is available in the client channel definition table (CCDT), the definition to use is randomly selected based on the weighting, with any applicable ClientChannelWeight(0) definitions selected first in alphabetical order.

Specify a value in the range 0 - 99. The default is 0.

A value of 0 indicates that no load balancing is performed and applicable definitions are selected in alphabetical order. To enable load balancing choose a value in the range 1 - 99 where 1 is the lowest weighting and 99 is the highest. The distribution of messages between two or more channels with non-zero weightings is proportional to the ratio of those weightings. For example, three channels with ClientChannelWeight values of 2, 4, and 14 are selected approximately 10%, 20%, and 70% of the time. This distribution is not guaranteed.

This attribute is valid for the client-connection channel type only.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_9.

ClusterPtr (MQPTR)

This field specifies the address a list of cluster names.

If *ClustersDefined* is greater than zero, this address is the address of a list of cluster names. The channel belongs to each cluster listed.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLUSSDR or MQCHT_CLUSRCVR.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_5.

ClustersDefined (MQLONG)

This field specifies the number of clusters to which the channel belongs.

This field is the number of cluster names pointed to by *ClusterPtr*. It is zero or greater.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLUSSDR or MQCHT_CLUSRCVR.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_5.

CLWLChannelPriority (MQLONG)

This field specifies the cluster workload channel priority.

The workload manager choose algorithm selects a destination with the highest priority from the set of destinations selected based on rank. If there are two possible destination queue managers, this attribute can be used to make one queue manager failover onto the other queue manager. All the messages go to the queue manager with the highest priority until that ends, then the messages go to the queue manager with the next highest priority.

The value is in the range 0 through 9. The default is 0.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_8.

For further information, see [Configuring a queue manager cluster](#).

CLWLChannelRank (MQLONG)

This field specifies the cluster workload channel rank.

The workload manager choose algorithm selects a destination with the highest rank. When the final destination is a queue manager on a different cluster, you can set the rank of intermediate gateway queue managers (at the intersection of neighboring clusters) so the choose algorithm correctly chooses a destination queue manager nearer the final destination.

The value is in the range 0 through 9. The default is 0.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_8.

For further information, see [Configuring a queue manager cluster](#).

CLWLChannelWeight (MQLONG)

This field specifies the cluster workload channel weight.

Cluster workload channel weight.

The workload manager choose algorithm uses the "weight" attribute of the channel to skew the destination choice so that more messages can be sent to a particular machine. For example, you can give a channel on a large UNIX server a larger "weight" than another channel on small desktop PC, and the choose algorithm chooses the UNIX server more frequently than the PC.

The value is in the range 1 through 99. The default is 50.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_8.

For further information, see [Configuring a queue manager cluster](#).

ConnectionAffinity (MQLONG)

This field specifies whether client applications that connect multiple times using the same queue manager name, use the same client channel.

Use this attribute when multiple applicable channel definitions are available.

The value is one of the following:

MQCAFTY_PREFERRED

The first connection in a process reading a client channel definition table (CCDT) creates a list of applicable definitions based on the weighting with any applicable CLNTWGHT(0) definitions first and in alphabetical order. Each connection in the process attempts to connect using the first definition in the list. If a connection is unsuccessful the next definition is used. Unsuccessful definitions with CLNTWGHT values other than 0 are moved to the end of the list. CLNTWGHT(0) definitions remain at the start of the list and are selected first for each connection.

Each client process with the same host name always creates the same list.

For client applications written in C, C++, or the .NET programming framework (including fully managed .NET) the list is updated if the CCDT has been modified since the list was created.

This value is the default value.

MQCAFTY_NONE

The first connection in a process reading a CCDT creates a list of applicable definitions. All connections in a process select an applicable definition based on the weighting with any applicable CLNTWGHT(0) definitions selected first in alphabetical order.

For client applications written in C, C++, or the .NET programming framework (including fully managed .NET) the list is updated if the CCDT has been modified since the list was created.

This attribute is valid for the client-connection channel type only.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_9.

ConnectionName (MQCHAR264)

This field specifies the connection name for the channel.

For cluster-receiver channels (when specified) CONNAME relates to the local queue manager, and for other channels it relates to the target queue manager. The value you specify depends on the transmission protocol (*TransportType*) to be used:

- For MQXPT_LU62, it is the fully-qualified name of the partner Logical Unit.
- For MQXPT_NETBIOS, it is the NetBIOS name defined on the remote machine.
- For MQXPT_TCP, it is either the host name, the network address of the remote machine specified in IPv4 dotted decimal or IPv6 hexadecimal format, or the local machine for cluster-receiver channels.
- For MQXPT_SPX, it is an SPX-style address comprising a 4 byte network address, a 6 byte node address, and a 2 byte socket number.

When defining a channel, this field is not relevant for channels with a *ChannelType* of MQCHT_SVRCONN or MQCHT_RECEIVER. However, when the channel definition is passed to an exit, this field contains the address of the partner, whatever the channel type.

The length of this field is given by MQ_CONN_NAME_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_2.

DataConversion (MQLONG)

This field specifies whether the sending message channel agent attempts conversion of the application message data if the receiving message channel agent is unable to perform this conversion.

This field applies only to messages that are not segments of logical messages; the MCA never attempts to convert messages which are segments.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR. It is one of the following:

MQCDC_SENDER_CONVERSION

Conversion by sender.

MQCDC_NO_SENDER_CONVERSION

No conversion by sender.

DefReconnect (MQLONG)

The DefReconnect channel attribute sets the default reconnection attribute value for a client connection channel.

The default automatic client reconnection option. You can configure a IBM MQ MQI client to automatically reconnect a client application. The IBM MQ MQI client tries to reconnect to a queue manager after a connection failure. It tries to reconnect without the application client issuing an MQCONN or MQCONNX MQI call.

Reconnection is an MQCONNX option. By using the DefReconnect channel attribute you can add reconnection behavior to existing applications that use MQCONN. You can also change the reconnection behavior of applications that use MQCONNX.

You can also set the DefRecon value from the mqclient.ini file to set or modify reconnection behavior. The DefRecon value from the mqclient.ini file takes precedence over the DefReconnect channel attribute.

Syntax

DefReconnect (MQRCN_NO | MQRCN_YES | MQRCN_Q_MGR |
MQRCN_DISABLED)

Parameters

MQRCN_NO

MQRCN_NO is the default value.

Unless overridden by MQCONNX, the client is not reconnected automatically.

MQRCN_YES

Unless overridden by MQCONNX, the client reconnects automatically.

MQRCN_Q_MGR

Unless overridden by MQCONNX, the client reconnects automatically, but only to the same queue manager. The QMGR option has the same effect as MQCNO_RECONNECT_Q_MGR.

MQRCN_DISABLED

Reconnection is disabled, even if requested by the client program using the MQCONNX MQI call.

Automatic client reconnection is not supported by IBM MQ classes for Java.

Table 643. Automatic reconnection depends on the values set in the application and in the channel definition

DefReconnect	Reconnection options set in the application			
	MQCNO_RECONNE CT	MQCNO_RECONNE CT_Q_MGR	MQCNO_RECONNE CT_AS_DEF	MQCNO_RECONNE CT_DISABLED
MQRCN_NO	YES	QMGR	NO	NO
MQRCN_YES	YES	QMGR	YES	NO
MQRCN_Q_MGR	YES	QMGR	QMGR	NO
MQRCN_DISABLED	NO	NO	NO	NO

Related concepts

[Automatic client reconnection](#)

[Channel and client reconnection](#)

[CHANNELS stanza of the client configuration file](#)

Related reference

[Connection options](#)

Options that control the action of MQCONNX.

Desc (MQCHAR64)

This field can be used for descriptive commentary.

The content of the field is of no significance to Message Channel Agents. However, it must contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the character set of the queue manager (as defined by the *CodedCharSetId* queue manager attribute), those characters might be translated incorrectly if this field is sent to another queue manager.

The length of this field is given by MQ_CHANNEL_DESC_LENGTH.

DiscInterval (MQLONG)

This field specifies the maximum time in seconds for which the channel waits for a message to arrive on the transmission queue, before terminating the channel.

In other words, it specifies the disconnect interval.

The A value of zero causes the MCA to wait indefinitely.

For server-connection channels using the TCP protocol, the interval represents the client inactivity disconnect value, specified in seconds. If a server-connection has received no communication from its partner client for this duration, it terminates the connection. The server-connection inactivity interval only applies between IBM MQ API calls from a client, so no client is disconnected during a long-running MQGET with wait call.

This attribute is not applicable for server-connection channels using protocols other than TCP.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, MQCHT_CLUSRCVR, or MQCHT_SVRCONN.

ExitDataLength (MQLONG)

This field specifies length in bytes of each of the user data items in the lists of exit user data items addressed by the *MsgUserDataPtr*, *SendUserDataPtr*, and *ReceiveUserDataPtr* fields.

This length is not necessarily the same as MQ_EXIT_DATA_LENGTH.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

ExitNameLength (MQLONG)

This field specifies the length in bytes of each of the names in the lists of exit names addressed by the *MsgExitPtr*, *SendExitPtr*, and *ReceiveExitPtr* fields.

This length is not necessarily the same as MQ_EXIT_NAME_LENGTH.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

HdrCompList [2] (MQLONG)

This field specifies the list of header data compression techniques which are supported by the channel.

The list contains one or more of the following values:

MQCOMPRESS_NONE

No header data compression is performed.

MQCOMPRESS_SYSTEM

Header data compression is performed.

MQCOMPRESS_NOT_AVAILABLE

Unused values in the list are set to this value.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_8.

HeartbeatInterval (MQLONG)

This field specifies the time in seconds between heartbeat flows.

The interpretation of this field depends on the channel type, as follows:

- For a channel type of MQCHT_SENDER, MQCHT_SERVER, MQCHT_RECEIVER, MQCHT_REQUESTER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR, this field is the time in seconds between heartbeat flows passed from the sending MCA when there are no messages on the transmission queue. This gives the receiving MCA the opportunity to quiesce the channel. To be useful, *HeartbeatInterval* must be less than *DiscInterval*.
- For a channel type of MQCHT_CLNTCONN or MQCHT_SVRCONN with the MQCD Sharing Conversations field set to zero, this field is the time in seconds between heartbeat flows passed from the server MCA when that MCA has issued an MQGET call with the MQGMO_WAIT option on behalf of a client application. This allows the server MCA to handle situations where the client connection fails during an MQGET with MQGMO_WAIT.
- For a channel type of MQCHT_CLNTCONN or MQCHT_SVRCONN with the MQCD Sharing Conversations field set to a non-zero value, this field is the time in seconds between heartbeat flow when there are no data flows sent or received. This allows the channel to be quiesced efficiently.

The value is in the range 0 through 999 999. The value that is used is the larger of the values specified at the sending side and receiving side unless a value of 0 is specified at either side, in which case no heartbeat exchange occurs.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

KeepAliveInterval (MQLONG)

This field specifies the value passed to the communications stack for keepalive timing for the channel.

The value is applicable for the TCP/IP and SPX communications protocols, though not all implementations support this parameter.

The value is in the range 0 through 99 999; the units are seconds. A value of zero indicates that channel keepalive is not enabled, although keepalive might still occur if TCP/IP keepalive (rather than channel keepalive) is enabled. The following special value is also valid:

MQKAI_AUTO

Automatic.

This value indicates that the keepalive interval is calculated from the negotiated heartbeat interval, as follows:

- If the negotiated heartbeat interval is greater than zero, the keepalive interval that is used is the heartbeat interval plus 60 seconds.
- If the negotiated heartbeat interval is zero, the keepalive interval that is used is zero.
- On z/OS, TCP/IP keepalive occurs when TCPKEEP(YES) is specified on the queue manager object.
- In other environments, TCP/IP keepalive occurs when the KEEPALIVE=YES parameter is specified in the TCP stanza in the distributed queuing configuration file.

This field is relevant only for channels that have a *TransportType* of MQXPT_TCP or MQXPT_SPX.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_7.

LocalAddress (MQCHAR48)

This field specifies the local TCP/IP address defined for the channel for outbound communications.

This field is blank if no specific address is defined for outbound communications. The address can optionally include a port number or range of port numbers. The format of this address is:

```
[ip-addr] [(low-port[,high-port])]
```

where square brackets ([]) denote optional information, ip-addr is specified in IPv4 dotted decimal, IPv6 hexadecimal, or alphanumeric form, and low-port and high-port are port numbers enclosed in parentheses. All are optional.

A specific IP address, port, or port range for outbound communications is useful in recovery scenarios where a channel is restarted on a different TCP/IP stack.

LocalAddress is similar in form to *ConnectionName*, but must not be confused with it. *LocalAddress* specifies the characteristics of the local communications, whereas *ConnectionName* specifies how to reach a remote queue manager.

This field is relevant only for channels with a *TransportType* of MQXPT_TCP, and a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_REQUESTER, MQCHT_CLNTCONN, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

The length of this field is given by MQ_LOCAL_ADDRESS_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_7.

LongMCAUserIdLength (MQLONG)

This field specifies the length in bytes of the full MCA user identifier pointed to by *LongMCAUserIdPtr*.

This field is not relevant for channels with a *ChannelType* of MQCHT_CLNTCONN.

This is an input/output field to the exit. The field is not present if *Version* is less than MQCD_VERSION_6.

LongMCAUserIdPtr (MQPTR)

This field specifies the address of the long MCA user identifier.

If *LongMCAUserIdLength* is greater than zero, this field is the address of the full MCA user identifier. The length of the full identifier is given by *LongMCAUserIdLength*. The first 12 bytes of the MCA user identifier are also contained in the field *MCAUserIdentifier*.

See the description of the *MCAUserIdentifier* field for details of the MCA user identifier.

This field is not relevant for channels with a *ChannelType* of MQCHT_SDR, MQCHT_SVR, MQCHT_CLNTCONN, or MQCHT_CLUSSDR.

This is an input/output field to the exit. The field is not present if *Version* is less than MQCD_VERSION_6.

LongRemoteUserIdLength (MQLONG)

This field specifies the length in bytes of the full remote user identifier pointed to by *LongRemoteUserIdPtr*.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLNTCONN or MQCHT_SVRCONN.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_6.

LongRemoteUserIdPtr (MQPTR)

This field specifies the address of the long remote user identifier.

If *LongRemoteUserIdLength* is greater than zero, this flag is the address of the full remote user identifier. The length of the full identifier is given by *LongRemoteUserIdLength*. The first 12 bytes of the remote user identifier are also contained in the field *RemoteUserIdentifier*.

See the description of the *RemoteUserIdentifier* field for details of the remote user identifier.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLNTCONN or MQCHT_SVRCONN.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_6.

LongRetryCount (MQLONG)

This field specifies the count used after the count specified by the *ShortRetryCount* has been exhausted.

It specifies the maximum number of further attempts that are made to connect to the remote machine, at intervals specified by *LongRetryInterval*, before logging an error to the operator.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

LongRetryInterval (MQLONG)

This field specifies the maximum number of seconds to wait before reattempting connection to the remote machine.

The interval between retries can be extended if the channel has to wait to become active.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

MaxInstances (MQLONG)

This field specifies the maximum number of simultaneous instances of an individual server-connection channel that can be started.

This field is used only on server-connection channels.

The field can have a value in the range 0 - 999 999 999. A value of zero prevents all client access.

The default value of this field is 999 999 999.

If the value of this field is reduced to a number that is less than the number of instances of the server-connection channel that are currently running, then those running instances are not affected. However, new instances cannot start until sufficient existing instances have ceased to run so that the number of currently running instances is less than the value of the field.

MaxInstancesPerClient (MQLONG)

This field specifies the maximum number of simultaneous instances of an individual server-connection channel that can be started from a single client.

In this context, connections that originate from the same remote network address are regarded as coming from the same client.

This field is used only on server-connection channels.

The field can have a value in the range 0 - 999 999 999. A value of zero prevents all client access.

The default value of this field is 999 999 999.

If the value of this field is reduced to a number that is less than the number of instances of the server-connection channel that are currently running from individual clients, then those running instances are not affected. However, new instances from any of those clients cannot start until sufficient existing instances have ceased to run such that the number of currently running instances, originating from the client attempting to start a new one, is less than the value of the field.

MaxMsgLength (MQLONG)

This field specifies the maximum message length that can be transmitted on the channel.

This is compared with the value for the remote channel and the actual maximum is the lower of the two values.

MCAName (MQCHAR20)

This field is a reserved field.

The value of this field is blank.

The length of this field is given by MQ_MCA_NAME_LENGTH.

MCASecurityId (MQBYTE40)

This field specifies the security identifier for the MCA.

This field is not relevant for channels with a *ChannelType* of MQCHT_CLNTCONN.

The following special value indicates that there is no security identifier:

MQSID_NONE

No security identifier specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQSID_NONE_ARRAY is also defined; this constant has the same value as MQSID_NONE, but is an array of characters instead of a string.

This is an input/output field to the exit. The length of this field is given by MQ_SECURITY_ID_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_6.

MCAType (MQLONG)

This field specifies the type of message channel agent program.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_REQUESTER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

The value is one of the following:

MQMCAT_PROCESS

Process.

The message channel agent runs as a separate process.

MQMCAT_THREAD

Thread (IBM i, UNIX, and Windows).

The message channel agent runs as a separate thread.

This field is not present when *Version* is less than MQCD_VERSION_2.

MCAUserIdentifier (MQCHAR12)

This field specifies the user identifier for the message channel agent (MCA).

This field uses the first 12 bytes of the MCA user identifier, and can be set by a security agent.

There are two fields that contain the MCA user identifier:

- *MCAUserIdentifier* contains the first 12 bytes of the MCA user identifier, and is padded with blanks if the identifier is shorter than 12 bytes. *MCAUserIdentifier* can be blank.
- *LongMCAUserIdPtr* points to the full MCA user identifier, which can be longer than 12 bytes. Its length is given by *LongMCAUserIdLength*. The full identifier contains no trailing blanks, and is not null-terminated. If the identifier is blank, *LongMCAUserIdLength* is zero, and the value of *LongMCAUserIdPtr* is undefined.

Note: *LongMCAUserIdPtr* is not present if *Version* is less than MQCD_VERSION_6.

If the MCA user identifier is nonblank, it specifies the user identifier to be used by the message channel agent for authorization to access IBM MQ resources. For channel types MQCHT_REQUESTER, MQCHT_RECEIVER, and MQCHT_CLUSRCVR, if PutAuthority is MQPA_DEFAULT this is the user identifier used for authorization checks for the put operation to destination queues.

If the MCA user identifier is blank, the message channel agent uses its default user identifier.

The MCA user identifier can be set by a security exit to indicate the user identifier that the message channel agent must use. The exit can change either *MCAUserIdentifier*, or the string pointed at by *LongMCAUserIdPtr*. If both are changed but differ from each other, the MCA uses *LongMCAUserIdPtr* in preference to *MCAUserIdentifier*. If the exit changes the length of the string addressed by *LongMCAUserIdPtr*, *LongMCAUserIdLength* must be set correspondingly. If the exit increases the length of the identifier, the exit must allocate storage of the required length, set that storage to the required identifier, and place the address of that storage in *LongMCAUserIdPtr*. The exit is responsible for freeing that storage when the exit is later invoked with the MQXR_TERM reason.

For channels with a *ChannelType* of MQCHT_SVRCONN, if *MCAUserIdentifier* in the channel definition is blank, any user identifier transferred from the client is copied into it. This user identifier (after any modification by the security exit at the server) is the one which the client application is assumed to be running under.

The MCA user identifier is not relevant for channels with a *ChannelType* of MQCHT_SDR, MQCHT_SVR, MQCHT_CLNTCONN, MQCHT_CLUSSDR.

This is an input/output field to the exit. The length of this field is given by MQ_USER_ID_LENGTH. This field is not present when *Version* is less than MQCD_VERSION_2.

ModeName (MQCHAR8)

This field specifies the LU 6.2 mode name.

This field is relevant only if the transmission protocol (*TransportType*) is MQXPT_LU62, and the *ChannelType* is not MQCHT_SVRCONN or MQCHT_RECEIVER.

This field is always blank. The information is contained in the communications Side Object instead.

The length of this field is given by MQ_MODE_NAME_LENGTH.

MsgCompList [16] (MQLONG)

This field specifies the list of message data compression techniques which are supported by the channel.

The list contains one or more of the following values:

MQCOMPRESS_NONE

No message data compression is performed.

MQCOMPRESS_RLE

Message data compression is performed using run-length encoding.

MQCOMPRESS_ZLIBFAST

Message data compression is performed using the zlib compression technique. A fast compression time is preferred.

MQCOMPRESS_ZLIBHIGH

Message data compression is performed using the zlib compression technique. A high level of compression is preferred.

MQCOMPRESS_ANY

Any compression technique supported by the queue manager can be used for message compression. MQCOMPRESS_ANY is only valid for receiver, requester, and server-connection channels.

MQCOMPRESS_NOT_AVAILABLE

Unused values in the list are set to this value.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_8.

MsgExit (MQCHARn)

This field specifies the channel message exit name.

If this name is nonblank, the exit is called at the following times:

- Immediately after a message has been retrieved from the transmission queue (sender or server), or immediately before a message is put to a destination queue (receiver or requester).

The exit is given the entire application message and transmission queue header for modification.

- At initialization and termination of the channel.

This field is not relevant for channels with a *ChannelType* of MQCHT_SVRCONN or MQCHT_CLNTCONN; a message exit is never invoked for such channels.

See “MQCD - Channel definition” on page 1487 for a description of the content of this field in various environments.

The length of this field is given by MQ_EXIT_NAME_LENGTH.

Note: The value of this constant is environment-specific.

MsgExitPtr (MQPTR)

This field specifies the address of the first *MsgExit* field.

If *MsgExitsDefined* is greater than zero, this address is the address of the list of names of each channel message exit in the chain.

Each name is in a field of length *ExitNameLength*, padded to the right with blanks. There are *MsgExitsDefined* fields adjoining one another - one for each exit.

Any changes made to these names by an exit are preserved, although the message channel exit takes no explicit action - it does not change which exits are invoked.

If *MsgExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

MsgExitsDefined (MQLONG)

This field specifies the number of channel message exits defined in the chain.

It is greater than or equal to zero.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

MsgRetryCount (MQLONG)

This field specifies the number of times MCA tries to put the message, after the first attempt has failed.

This field indicates the number of times that the MCA tries the open or put operation, if the first MQOPEN or MQPUT fails with completion code MQCC_FAILED. The effect of this attribute depends on whether *MsgRetryExit* is blank or nonblank:

- If *MsgRetryExit* is blank, the *MsgRetryCount* attribute controls whether the MCA attempts retries. If the attribute value is zero, no retries are attempted. If the attribute value is greater than zero, the retries are attempted at intervals given by the *MsgRetryInterval* attribute.

Retries are attempted only for the following reason codes:

- MQRC_PAGESET_FULL
- MQRC_PUT_INHIBITED
- MQRC_Q_FULL

For other reason codes, the MCA proceeds immediately to its normal failure processing, without retrying the failing message.

- If *MsgRetryExit* is nonblank, the *MsgRetryCount* attribute does not affect the MCA; instead it is the message-retry exit which determines how many times the retry is attempted, and at what intervals; the exit is invoked even if the *MsgRetryCount* attribute is zero.

The *MsgRetryCount* attribute is made available to the exit in the MQCD structure, but the exit is not required to honor it - retries continue indefinitely until the exit returns MQXCC_SUPPRESS_FUNCTION in the *ExitResponse* field of MQCXP.

This field is relevant only for channels with a *ChannelType* of MQCHT_REQUESTER, MQCHT_RECEIVER, or MQCHT_CLUSRCVR.

This field is not present when *Version* is less than MQCD_VERSION_3.

MsgRetryExit (MQCHARn)

This field specifies the channel message retry exit name.

The message retry exit is an exit that is invoked by the MCA when the MCA receives a completion code of MQCC_FAILED from an MQOPEN or MQPUT call. The purpose of the exit is to specify a time interval for which the MCA waits before trying the MQOPEN or MQPUT operation again. Alternatively, the exit can be set to not try the operation again.

The exit is invoked for all reason codes that have a completion code of MQCC_FAILED - the settings of the exit determine which reason codes it wants the MCA to try again, for how many attempts, and at what time intervals.

When the operation is not to be attempted any more, the MCA performs its normal failure processing; this processing includes generating an exception report message (if specified by the sender), and either placing the original message on the dead-letter queue or discarding the message (according to whether the sender specified MQRO_DEAD_LETTER_Q or MQRO_DISCARD_MSG). Failures involving the dead-letter queue (for example, dead-letter queue full) do not cause the message-retry exit to be invoked.

If the exit name is nonblank, the exit is called at the following times:

- Immediately before performing the wait before trying to deliver a message again
- At initialization and termination of the channel

See “MQCD - Channel definition” on page 1487 for a description of the content of this field in various environments.

This field is relevant only for channels with a *ChannelType* of MQCHT_REQUESTER, MQCHT_RECEIVER, or MQCHT_CLUSRCVR.

The length of this field is given by MQ_EXIT_NAME_LENGTH.

Note: The value of this constant is environment-specific.

This field is not present when *Version* is less than MQCD_VERSION_3.

MsgRetryInterval (MQLONG)

This field specifies the minimum interval in milliseconds after which the open or put operation is retried.

The effect of this attribute depends on whether *MsgRetryExit* is blank or nonblank:

- If *MsgRetryExit* is blank, the *MsgRetryInterval* attribute specifies the minimum period that the MCA waits before retrying a message, if the first MQOPEN or MQPUT fails with completion code MQCC_FAILED. A value of zero means that the retry will be performed as soon as possible after the previous attempt. Retries are performed only if *MsgRetryCount* is greater than zero.

This attribute is also used as the wait time if the message-retry exit returns an invalid value in the *MsgRetryInterval* field in MQCXP.

- If *MsgRetryExit* is not blank, the *MsgRetryInterval* attribute does not affect the MCA; instead it is the message-retry exit which determines how long the MCA waits. The *MsgRetryInterval* attribute is made available to the exit in the MQCD structure, but the exit is not required to honor it.

The value is in the range 0 through 999 999 999.

This field is relevant only for channels with a *ChannelType* of MQCHT_REQUESTER, MQCHT_RECEIVER, or MQCHT_CLUSRCVR.

This field is not present when *Version* is less than MQCD_VERSION_3.

The following fields in this structure are not present if *Version* is less than MQCD_VERSION_4.

MsgRetryUserData (MQCHAR32)

This field specifies the channel message retry exit user data.

This data is passed to the channel message-retry exit in the *ExitData* field of the *ChannelExitParms* parameter (see MQ_CHANNEL_EXIT).

This field initially contains the data that was set in the channel definition. However, during the lifetime of this MCA instance, any changes made to the contents of this field by an exit of any type are preserved by the MCA, and made visible to subsequent invocations of exits (regardless of type) for this MCA instance. Such changes do not affect the channel definition used by other MCA instances. Any characters (including binary data) can be used.

This field is relevant only for channels with a *ChannelType* of MQCHT_REQUESTER, MQCHT_RECEIVER, or MQCHT_CLUSRCVR.

The length of this field is given by MQ_EXIT_DATA_LENGTH. This field is not present when *Version* is less than MQCD_VERSION_3.

This field is not relevant in IBM MQ for IBM i.

MsgUserData (MQCHAR32)

This field specifies channel message exit user data.

This data is passed to the channel message exit in the *ExitData* field of the *ChannelExitParms* parameter (see MQ_CHANNEL_EXIT).

This field initially contains the data that was set in the channel definition. However, during the lifetime of this MCA instance, any changes made to the contents of this field by an exit of any type are preserved by the MCA, and made visible to subsequent invocations of exits (regardless of type) for this MCA instance. Such changes do not affect the channel definition used by other MCA instances. Any characters (including binary data) can be used.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

This field is not relevant in IBM MQ for IBM i.

MsgUserDataPtr (MQPTR)

This field specifies the address of the first *MsgUserData* field.

If *MsgExitsDefined* is greater than zero, this address is the address of the list of user data items for each channel message exit in the chain.

Each user data item is in a field of length *ExitDataLength*, padded to the right with blanks. There are *MsgExitsDefined* fields adjoining one another - one for each exit. If the number of user data items defined is less than the number of exit names, undefined user data items are set to blanks. Conversely, if the number of user data items defined is greater than the number of exit names, the excess user data items are ignored and not presented to the exit.

Any changes made to these values by an exit are preserved. This allows one exit to pass information to another exit. No validation is carried out on any changes so, for example, binary data can be written to these fields if required.

If *MsgExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

NetworkPriority (MQLONG)

This field specifies the priority of the network connection for the channel.

When multiple paths to a particular destination are available, the path with the highest priority is chosen. The value is in the range 0 through 9; 0 is the lowest priority.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLUSSDR or MQCHT_CLUSRCVR.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_5.

The following fields in this structure are not present if *Version* is less than MQCD_VERSION_6.

NonPersistentMsgSpeed (MQLONG)

This field specifies the speed at which nonpersistent messages travel through the channel.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_RECEIVER, MQCHT_REQUESTER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

The value is one of the following:

MQNPMS_NORMAL

Normal speed.

If a channel is defined to be MQNPMS_NORMAL, nonpersistent messages travel through the channel at normal speed. This has the advantage that these messages are not lost if there is a channel failure. Also, persistent and nonpersistent messages on the same transmission queue maintain their order relative to each other.

MQNPMS_FAST

Fast speed.

If a channel is defined to be MQNPMS_FAST, nonpersistent messages travel through the channel at fast speed. This improves the throughput of the channel, but means that nonpersistent messages are lost if there is a channel failure. Also, it is possible for nonpersistent messages to jump ahead of persistent messages waiting on the same transmission queue, that is, the order of nonpersistent messages is not maintained relative to persistent messages. However the order of nonpersistent messages relative to each other is maintained. Similarly, the order of persistent messages relative to each other is maintained.

Password (MQCHAR12)

This field specifies the password used by the message channel agent when attempting to initiate a secure SNA session with a remote message channel agent.

This field can be nonblank only on UNIX systems, and Windows, and is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_REQUESTER, or MQCHT_CLNTCONN. On z/OS, this field is not relevant.

The length of this field is given by MQ_PASSWORD_LENGTH. However, only the first 10 characters are used.

This field is not present if *Version* is less than MQCD_VERSION_2.

PropertyControl (MQLONG)

This field specifies what happens to properties of messages when the message is about to be sent to a V6 or prior queue manager (a queue manager that does not understand the concept of a property descriptor).

The value can be any of the following values:

MQPROP_COMPATIBILITY

If the message contains a property with a prefix of **mcd.**, **jms.**, **usr.**, or **mqext.**, all message properties are delivered to the application in an MQRFH2 header. Otherwise all properties of the message, except those properties contained in the message descriptor (or extension), are discarded and are no longer accessible to the application.

This value is the default value; it allows applications, which expect JMS-related properties to be in an MQRFH2 header in the message data, to continue to work unmodified.

MQPROP_NONE

All properties of the message, except those properties in the message descriptor (or extension), are removed from the message before the message is sent to the remote queue manager.

MQPROP_ALL

All properties of the message are included with the message when it is sent to the remote queue manager. The properties, except those properties in the message descriptor (or extension), are placed in one or more MQRFH2 headers in the message data.

This attribute is applicable to Sender, Server, Cluster Sender, and Cluster Receiver channels.

[“MQIA_* \(Integer Attribute Selectors\)” on page 125](#)

[“MQPROP_* \(Queue and Channel Property Control Values and Maximum Properties Length\)” on page 165](#)

PutAuthority (MQLONG)

This field specifies whether the user identifier in the context information associated with a message is used to establish authority to put the message to the destination queue.

This field is relevant only for channels with a *ChannelType* of MQCHT_REQUESTER, MQCHT_RECEIVER, or MQCHT_CLUSRCVR. It is one of the following:

MQPA_DEFAULT

Default user identifier is used.

MQPA_CONTEXT

Context user identifier is used.

MQPA_ALTERNATE_OR_MCA

The user ID from the UserIdentifier field of the message descriptor is used. Any user ID received from the network is not used. This value is supported only on z/OS.

MQPA_ONLY_MCA

The default user ID is used. Any user ID received from the network is not used. This value is supported only on z/OS.

QMgrName (MQCHAR48)

This field specifies the name of the queue manager that an exit can connect to.

For channels with a *ChannelType* other than MQCHT_CLNTCONN, this field is the name of the queue manager that an exit can connect to, which on UNIX, Linux and Windows systems, is always nonblank.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH.

ReceiveExit (MQCHARn)

This field specifies the channel receive exit name.

If this name is nonblank, the exit is called at the following times:

- Immediately before the received network data is processed.

The exit is given the complete transmission buffer as received. The contents of the buffer can be modified as required.

- At initialization and termination of the channel.

See [“MQCD - Channel definition” on page 1487](#) for a description of the content of this field in various environments.

The length of this field is given by MQ_EXIT_NAME_LENGTH.

Note: The value of this constant is environment-specific.

ReceiveExitPtr (MQPTR)

This field specifies the address of the first *ReceiveExit* field.

If *ReceiveExitsDefined* is greater than zero, this address is the address of the list of names of each channel receive exit in the chain.

Each name is in a field of length *ExitNameLength*, padded to the right with blanks. There are *ReceiveExitsDefined* fields adjoining one another - one for each exit.

Any changes made to these names by an exit are preserved, although the message channel exit takes no explicit action - it does not change which exits are invoked.

If *ReceiveExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

ReceiveExitsDefined (MQLONG)

This field specifies the number of channel receive exits defined in the chain.

It is greater than or equal to zero.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

ReceiveUserData (MQCHAR32)

This channel specifies channel receive exit user data.

This data is passed to the channel receive exit in the *ExitData* field of the *ChannelExitParms* parameter (see MQ_CHANNEL_EXIT).

This field initially contains the data that was set in the channel definition. However, during the lifetime of this MCA instance, any changes made to the contents of this field by an exit of any type are preserved by the MCA, and made visible to subsequent invocations of exits (regardless of type) for this MCA instance. This applies to exits on different conversations. Such changes do not affect the channel definition used by other MCA instances. Any characters (including binary data) can be used.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

This field is not relevant in IBM MQ for IBM i.

The following fields in this structure are not present if *Version* is less than MQCD_VERSION_2.

ReceiveUserDataPtr (MQPTR)

This field specifies the address of the first *ReceiveUserData* field.

If *ReceiveExitsDefined* is greater than zero, this address is the address of the list of user data item for each channel receive exit in the chain.

Each user data item is in a field of length *ExitDataLength*, padded to the right with blanks. There are *ReceiveExitsDefined* fields adjoining one another - one for each exit. If the number of user data items defined is less than the number of exit names, undefined user data items are set to blanks. Conversely, if the number of user data items defined is greater than the number of exit names, the excess user data items are ignored and not presented to the exit.

Any changes made to these values by an exit are preserved. This allows one exit to pass information to another exit. No validation is carried out on any changes so, for example, binary data can be written to these fields if required.

If *ReceiveExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

The following fields in this structure are not present if *Version* is less than MQCD_VERSION_5.

RemotePassword (MQCHAR12)

This field specifies the password from a partner.

This field contains valid information only if *ChannelType* is MQCHT_CLNTCONN or MQCHT_SVRCONN.

- For a security exit at an MQCHT_CLNTCONN channel, this password is a password which has been obtained from the environment. The exit can choose to send it to the security exit at the server.
- For a security exit at an MQCHT_SVRCONN channel, this field might contain a password which has been obtained from the environment at the client, if there is no client security exit. The exit can use this password to validate the user identifier in *RemoteUserIdentifier*.

If there is a security exit at the client, then this information can be obtained in a security flow from the client.

The length of this field is given by MQ_PASSWORD_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_2.

RemoteSecurityId (MQBYTE40)

This field specifies the security identifier for the remote user.

This field is relevant only for channels with a *ChannelType* of MQCHT_CLNTCONN or MQCHT_SVRCONN.

The following special value indicates that there is no security identifier:

MQSID_NONE

No security identifier specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQSID_NONE_ARRAY is also defined; this constant has the same value as MQSID_NONE, but is an array of characters instead of a string.

This is an input field to the exit. The length of this field is given by MQ_SECURITY_ID_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_6.

The following fields in this structure are not present if *Version* is less than MQCD_VERSION_7.

RemoteUserIdentifier (MQCHAR12)

This field specifies the first 12 bytes of a user identifier from a partner.

There are two fields that contain the remote user identifier:

- *RemoteUserIdentifier* contains the first 12 bytes of the remote user identifier, and is padded with blanks if the identifier is shorter than 12 bytes. *RemoteUserIdentifier* can be blank.
- *LongRemoteUserIdPtr* points to the full remote user identifier, which can be longer than 12 bytes. Its length is given by *LongRemoteUserIdLength*. The full identifier contains no trailing blanks, and is not null-terminated. If the identifier is blank, *LongRemoteUserIdLength* is zero, and the value of *LongRemoteUserIdPtr* is undefined.

LongRemoteUserIdPtr is not present if *Version* is less than MQCD_VERSION_6.

The remote user identifier is relevant only for channels with a *ChannelType* of MQCHT_CLNTCONN or MQCHT_SVRCONN.

- For a security exit on an MQCHT_CLNTCONN channel, this value is a user identifier that has been obtained from the environment. The exit can choose to send it to the security exit at the server.
- For a security exit on an MQCHT_SVRCONN channel, this field might contain a user identifier which has been obtained from the environment at the client, if there is no client security exit. The exit might validate this user ID (possibly with the password in *RemotePassword*) and update the value in *MCAUserIdentifier*.

If there is a security exit at the client, then this information can be obtained in a security flow from the client.

The length of this field is given by MQ_USER_ID_LENGTH. This field is not present if *Version* is less than MQCD_VERSION_2.

SecurityExit (MQCHARn)

This field specifies the channel security exit name.

If this name is nonblank, the exit is called at the following times:

- Immediately after establishing a channel.

Before any messages are transferred, the exit is given the opportunity to instigate security flows to validate connection authorization.

- Upon receipt of a response to a security message flow.

Any security message flows received from the remote processor on the remote machine are given to the exit.

- At initialization and termination of the channel.

See [“MQCD - Channel definition” on page 1487](#) for a description of the content of this field in various environments.

The length of this field is given by MQ_EXIT_NAME_LENGTH.

Note: The value of this constant is environment-specific.

SecurityUserData (MQCHAR32)

This channel specifies the channel security exit user data.

This data is passed to the channel security exit in the *ExitData* field of the *ChannelExitParms* parameter (see MQ_CHANNEL_EXIT).

This field initially contains the data that was set in the channel definition. However, during the lifetime of this MCA instance, any changes made to the contents of this field by an exit of any type are preserved by the MCA, and made visible to subsequent invocations of exits (regardless of type) for this MCA instance. This applies to exits on different conversations. Such changes do not effect on the channel definition used by other MCA instances. Any characters (including binary data) can be used.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

This field is not relevant in IBM MQ for IBM i.

SendExit (MQCHARn)

This field specifies the channel send exit name.

If this name is nonblank, the exit is called at the following times:

- Immediately before data is sent out on the network.

The exit is given the complete transmission buffer before it is transmitted. The contents of the buffer can be modified as required.

- At initialization and termination of the channel.

See [“MQCD - Channel definition” on page 1487](#) for a description of the content of this field in various environments.

The length of this field is given by MQ_EXIT_NAME_LENGTH.

Note: The value of this constant is environment-specific.

SendExitPtr (MQPTR)

This field specifies the address of the first *SendExit* field.

If *SendExitsDefined* is greater than zero, this address is the address of the list of names of each channel send exit in the chain.

Each name is in a field of length *ExitNameLength*, padded to the right with blanks. There are *SendExitsDefined* fields adjoining one another - one for each exit.

Any changes made to these names by an exit are preserved, although the message send exit takes no explicit action - it does not change which exits are invoked.

If *SendExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

SendExitsDefined (MQLONG)

This field specifies the number of channel send exits defined in the chain.

It is greater than or equal to zero.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

SendUserData (MQCHAR32)

This field specifies the channel send exit user data.

This data is passed to the channel send exit in the *ExitData* field of the *ChannelExitParms* parameter (see MQ_CHANNEL_EXIT).

This field initially contains the data that was set in the channel definition. However, during the lifetime of this MCA instance, any changes made to the contents of this field by an exit of any type are preserved by the MCA, and made visible to subsequent invocations of exits (regardless of type) for this MCA instance. This applies to exits on different conversations. Such changes do not affect the channel definition used by other MCA instances. Any characters (including binary data) can be used.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

This field is not relevant in IBM MQ for IBM i.

SendUserDataPtr (MQPTR)

This field specifies the address of the *SendUserData* field.

If *SendExitsDefined* is greater than zero, this address is the address of the list of user data items for each channel message exit in the chain.

Each user data item is in a field of length *ExitDataLength*, padded to the right with blanks. There are *MsgExitsDefined* fields adjoining one another - one for each exit. If the number of user data items defined is less than the number of exit names, undefined user data items are set to blanks. Conversely, if the number of user data items defined is greater than the number of exit names, the excess user data items are ignored and not presented to the exit.

Any changes made to these values by an exit are preserved. This allows one exit to pass information to another exit. No validation is carried out on any changes so, for example, binary data can be written to these fields if required.

If *SendExitsDefined* is zero, this field is the null pointer.

On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_4.

SeqNumberWrap (MQLONG)

This field specifies the highest allowable message sequence number.

When this value is reached, sequence numbers wrap to start again at 1.

This value is non-negotiable and must match in both the local and remote channel definitions.

This field is not relevant for channels with a *ChannelType* of MQCHT_SVRCONN or MQCHT_CLNTCONN.

SharingConversations (MQLONG)

This field specifies the maximum number of conversations that can share a channel instance associated with this channel.

This field is used on client connection and server-connection channels.

A value of 0 means that the channel operates as it did in versions earlier than IBM WebSphere MQ 7.0 with respect to the following attributes:

- Conversation sharing
- Read ahead
- STOP CHANNEL(<channelname>) MODE(QUIESCE)
- Heartbeating
- Client asynchronous consumption

A value of 1 is the minimum value for IBM MQ V7.0 behavior. Although only one conversation is allowed on the channel instance, read ahead, asynchronous consumption, and the Version 7 behavior of CLNTCONN - SVRCONN heartbeating and quiescent channel stopping are available.

This is an input field to the exit. It is not present if *Version* is less than MQCD_VERSION_9.

The default value of this field is 10.

Note: *MaxInstances* and *MaxInstancesPerClient* limits applied to a channel restrict the number of channel instances, not the number of conversations that might be sharing those instances.

ShortConnectionName (MQCHAR20)

This field specifies the first 20 bytes of a connection name.

If the *Version* field is MQCD_VERSION_1, *ShortConnectionName* contains the full connection name.

If the *Version* field is MQCD_VERSION_2 or greater, *ShortConnectionName* contains the first 20 characters of the connection name. The full connection name is given by the *ConnectionName* field; *ShortConnectionName* and the first 20 characters of *ConnectionName* are identical.

See *ConnectionName* for details of the contents of this field.

Note: The name of this field was changed for MQCD_VERSION_2 and subsequent versions of MQCD; the field was previously called *ConnectionName*.

The length of this field is given by MQ_SHORT_CONN_NAME_LENGTH.

ShortRetryCount (MQLONG)

This field specifies the maximum number of attempts that are made to connect to a remote machine.

This field is the maximum number of attempts that are made to connect to the remote machine, at intervals specified by *ShortRetryInterval*, before the (normally longer) *LongRetryCount* and *LongRetryInterval* are used.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

ShortRetryInterval (MQLONG)

This field specifies the maximum number of seconds to wait before reattempting connection to the remote machine.

The interval between retries might be extended if the channel has to wait to become active.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_CLUSSDR, or MQCHT_CLUSRCVR.

SSLCipherSpec (MQCHAR32)

This field specifies the Cipher Spec that is in use when using SSL.

If SSLCipherSpec is blank, the channel is not using SSL. If it is not blank, this field contains a string specifying the CipherSpec in use.

This parameter is valid for all channel types. It is supported on AIX, HP-UX, Linux, IBM i, Solaris, Windows, and z/OS. It is valid only for channel types of a transport type (TRPTYPE) of TCP.

This is an input field to the exit. The length of this field is given by MQ_SSL_CIPHER_SPEC_LENGTH. The field is not present if *Version* is less than MQCD_VERSION_7.

SSLClientAuth (MQLONG)

This field specifies whether SSL client authentication is required.

This field is relevant only to SVRCONN channel definitions.

It is one of the following values:

MQSCA_REQUIRED

Client authentication required.

MQSCA_OPTIONAL

Client authentication optional.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_7.

SSLPeerNameLength (MQLONG)

This field specifies the length in bytes of the SSL peer name pointed to by *SSLPeerNamePtr*.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_7.

SSLPeerNamePtr (MQPTR)

This field specifies the address of the SSL peer name.

When a certificate is received during a successful SSL handshake, the Distinguished Name of the subject of the certificate is copied into the MQCD field accessed by *SSLPeerNamePtr* at the end of the channel which receives the certificate. It overwrites the *SSLPeerName* value for the channel if this value is present in the channel definition of the local user. If a security exit is specified at this end of the channel it receives the Distinguished Name from the peer certificate in the MQCD.

This is an input field to the exit. The field is not present if *Version* is less than MQCD_VERSION_7.

Note: Security exit applications constructed prior to the release of IBM MQ v7.1 may require updating. For more information see [Channel security exit programs](#).

StrucLength (MQLONG)

This field specifies the length in bytes of the MQCD structure.

The length does not include any of the strings addressed by pointer fields contained within the structure. The value is one of the following:

MQCD_LENGTH_4

Length of version-4 channel definition structure.

MQCD_LENGTH_5

Length of version-5 channel definition structure.

MQCD_LENGTH_6

Length of version-6 channel definition structure.

MQCD_LENGTH_7

Length of version-7 channel definition structure.

MQCD_LENGTH_8

Length of version-8 channel definition structure.

MQCD_LENGTH_9

Length of version-9 channel definition structure.

The following constant specifies the length of the current version:

MQCD_CURRENT_LENGTH

Length of current version of channel definition structure.

Note: These constants have values that are environment-specific.

The field is not present if *Version* is less than MQCD_VERSION_4.

TpName (MQCHAR64)

This field specifies the LU 6.2 transaction program name.

This field is relevant only if the transmission protocol (*TransportType*) is MQXPT_LU62, and the *ChannelType* is not MQCHT_SVRCONN or MQCHT_RECEIVER.

This field is always blank on platforms on which the information is contained in the communications Side Object instead.

The length of this field is given by MQ_TP_NAME_LENGTH.

TransportType (MQLONG)

This field specifies the transmission protocol to be used.

The value is not checked if the channel was initiated from the other end.

It is one of the following values:

MQXPT_LU62

LU 6.2 transport protocol.

MQXPT_TCP

TCP/IP transport protocol.

MQXPT_NETBIOS

NetBIOS transport protocol.

This value is supported in the following environments: Windows.

MQXPT_SPX

SPX transport protocol.

This value is supported in the following environments: Windows, plus IBM MQ clients connected to these systems.

UseDLQ (MQLONG)

This field specifies whether the dead-letter queue (or undelivered message queue) is used when messages cannot be delivered by channels.

It can contain one of the following values:

MQUSEDLQ_NO

Messages that cannot be delivered by a channel are treated as a failure. The channel either discards the message, or the channel ends, in accordance with the NPMSPPEED setting.

MQUSEDLQ_YES

When the DEADQ queue manager attribute provides the name of a dead-letter queue, then it is used, else the behavior is as for NO. YES is the default value.

UserIdentifier (MQCHAR12)

This field specifies the user identifier used by the message channel agent when attempting to initiate a secure SNA session with a remote message channel agent.

This field can be nonblank only on UNIX systems and Windows, and is relevant only for channels with a *ChannelType* of MQCHT_SENDER, MQCHT_SERVER, MQCHT_REQUESTER, or MQCHT_CLNTCONN. On z/OS, this field is not relevant.

The length of this field is given by MQ_USER_ID_LENGTH. However, only the first 10 characters are used.

This field is not present when *Version* is less than MQCD_VERSION_2.

Version (MQLONG)

The *Version* field specifies the highest version number that you can set for the structure.

The value depends on the environment:

MQCD _VERSION_1

Version 1 channel definition structure.

MQCD _VERSION_2

Version 2 channel definition structure.

MQCD _VERSION_3

Version 3 channel definition structure.

MQCD _VERSION_4

Version 4 channel definition structure.

MQCD _VERSION_5

Version 5 channel definition structure.

MQCD _VERSION_6

Version 6 channel definition structure.

MQCD _VERSION_7

Version 7 channel definition structure.

MQCD _VERSION_8

Version 8 channel definition structure.

MQCD _VERSION_9

Version 9 channel definition structure.

Version 9 is the highest that you can set the field to on IBM WebSphere MQ 7.0 and IBM WebSphere MQ 7.0.1 on all platforms.

MQCD _VERSION_10

Version 10 channel definition structure.

Version 10 is the highest that you can set the field to on IBM WebSphere MQ 7.1 and IBM WebSphere MQ 7.5 on all platforms.

MQCD _VERSION_11

Version 11 channel definition structure.

Version 11 is the highest that you can set the field to on IBM MQ 8.0 on all platforms.

Fields that exist only in the more recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQCD_CURRENT_VERSION

The value set in MQCD_CURRENT_VERSION is the current version of the channel definition structure being used.

The value of MQCD_CURRENT_VERSION depends on the environment. It contains the highest value supported by the platform.

MQCD_CURRENT_VERSION is not used to initialize the default structures provided in the header, copy, and include files provided for different programming languages. The default initialization of Version depends on the platform and release.

For IBM WebSphere MQ 7.0 and later versions, the MQCD declarations in the header, copy, and include files are initialized to MQCD_VERSION_6. To use additional MQCD fields, applications must set the version number to MQCD_CURRENT_VERSION. If you are writing an application that is portable between several environments, you must choose a version that is supported in all the environments.

Tip: When a new version of the MQCD structure is introduced, the layout of the existing part is not changed. The exit must check the version number. It must be equal to or greater than the lowest version that contains the fields that the exit needs to use.

XmitQName (MQCHAR48)

This field specifies the name of the transmission queue from which messages are retrieved.

This field is relevant only for channels with a *ChannelType* of MQCHT_SENDER or MQCHT_SERVER.

The length of this field is given by MQ_Q_NAME_LENGTH.

C declaration

This declaration is the C declaration for the MQCD structure.

```
typedef struct tagMQCD MQCD;
typedef MQCD MQPOINTER PMQCD;
typedef PMQCD MQPOINTER PPMQCD;

struct tagMQCD {
    MQCHAR    ChannelName[20];          /* Channel definition name */
    MQLONG    Version;                  /* Structure version number */
    MQLONG    ChannelType;              /* Channel type */
    MQLONG    TransportType;            /* Transport type */
    MQCHAR    Desc[64];                 /* Channel description */
    MQCHAR    QMgrName[48];             /* Queue-manager name */
    MQCHAR    XmitQName[48];            /* Transmission queue name */
    MQCHAR    ShortConnectionName[20];  /* First 20 bytes of */
                                        /* connection name */
    MQCHAR    MCAName[20];              /* Reserved */
    MQCHAR    ModeName[8];              /* LU 6.2 Mode name */
    MQCHAR    TpName[64];               /* LU 6.2 transaction program */
                                        /* name */
    MQLONG    BatchSize;                /* Batch size */
    MQLONG    DiscInterval;             /* Disconnect interval */
    MQLONG    ShortRetryCount;          /* Short retry count */
    MQLONG    ShortRetryInterval;       /* Short retry wait interval */
    MQLONG    LongRetryCount;           /* Long retry count */
    MQLONG    LongRetryInterval;        /* Long retry wait interval */
    MQCHAR    SecurityExit[128];        /* Channel security exit name */
    MQCHAR    MsgExit[128];             /* Channel message exit name */
    MQCHAR    SendExit[128];            /* Channel send exit name */
    MQCHAR    ReceiveExit[128];         /* Channel receive exit name */
    MQLONG    SeqNumberWrap;            /* Highest allowable message */
                                        /* sequence number */
    MQLONG    MaxMsgLength;             /* Maximum message length */
    MQLONG    PutAuthority;             /* Put authority */
    MQLONG    DataConversion;           /* Data conversion */
    MQCHAR    SecurityUserData[32];     /* Channel security exit user */
}
```

```

MQCHAR    MsgUserData[32];          /* data */
MQCHAR    SendUserData[32];         /* Channel message exit user */
MQCHAR    ReceiveUserData[32];      /* data */
MQCHAR    ReceiveUserData[32];      /* Channel receive exit user */
MQCHAR    ReceiveUserData[32];      /* data */

/* Ver:1 */
MQCHAR    UserIdentifier[12];        /* User identifier */
MQCHAR    Password[12];             /* Password */
MQCHAR    MCAUserIdentifier[12];     /* First 12 bytes of MCA user */
MQCHAR    MCAUserIdentifier[12];     /* identifier */
MQLONG    MCAType;                  /* Message channel agent type */
MQCHAR    ConnectionName[264];       /* Connection name */
MQCHAR    RemoteUserIdentifier[12];  /* First 12 bytes of user */
MQCHAR    RemoteUserIdentifier[12];  /* identifier from partner */
MQCHAR    RemotePassword[12];        /* Password from partner */

/* Ver:2 */
MQCHAR    MsgRetryExit[128];         /* Channel message retry exit */
MQCHAR    MsgRetryExit[128];         /* name */
MQCHAR    MsgRetryUserData[32];      /* Channel message retry exit */
MQCHAR    MsgRetryUserData[32];      /* user data */
MQLONG    MsgRetryCount;             /* Number of times MCA will */
MQLONG    MsgRetryCount;             /* try to put the message, */
MQLONG    MsgRetryCount;             /* after first attempt has */
MQLONG    MsgRetryCount;             /* failed */
MQLONG    MsgRetryInterval;          /* Minimum interval in */
MQLONG    MsgRetryInterval;          /* milliseconds after which */
MQLONG    MsgRetryInterval;          /* the open or put operation */
MQLONG    MsgRetryInterval;          /* will be retried */

/* Ver:3 */
MQLONG    HeartbeatInterval;         /* Time in seconds between */
MQLONG    HeartbeatInterval;         /* heartbeat flows */
MQLONG    BatchInterval;             /* Batch duration */
MQLONG    NonPersistentMsgSpeed;     /* Speed at which */
MQLONG    NonPersistentMsgSpeed;     /* nonpersistent messages are */
MQLONG    NonPersistentMsgSpeed;     /* sent */
MQLONG    StrucLength;               /* Length of MQCD structure */
MQLONG    ExitNameLength;            /* Length of exit name */
MQLONG    ExitDataLength;            /* Length of exit user data */
MQLONG    MsgExitsDefined;           /* Number of message exits */
MQLONG    MsgExitsDefined;           /* defined */
MQLONG    SendExitsDefined;           /* Number of send exits */
MQLONG    SendExitsDefined;           /* defined */
MQLONG    ReceiveExitsDefined;        /* Number of receive exits */
MQLONG    ReceiveExitsDefined;        /* defined */
MQPTR     MsgExitPtr;                /* Address of first MsgExit */
MQPTR     MsgExitPtr;                /* field */
MQPTR     MsgUserDataPtr;            /* Address of first */
MQPTR     MsgUserDataPtr;            /* MsgUserData field */
MQPTR     SendExitPtr;               /* Address of first SendExit */
MQPTR     SendExitPtr;               /* field */
MQPTR     SendUserDataPtr;           /* Address of first */
MQPTR     SendUserDataPtr;           /* SendUserData field */
MQPTR     ReceiveExitPtr;            /* Address of first */
MQPTR     ReceiveExitPtr;            /* ReceiveExit field */
MQPTR     ReceiveUserDataPtr;        /* Address of first */
MQPTR     ReceiveUserDataPtr;        /* ReceiveUserData field */

/* Ver:4 */
MQPTR     ClusterPtr;                /* Address of a list of */
MQPTR     ClusterPtr;                /* cluster names */
MQLONG    ClustersDefined;           /* Number of clusters to */
MQLONG    ClustersDefined;           /* which the channel belongs */
MQLONG    NetworkPriority;           /* Network priority */

/* Ver:5 */
MQLONG    LongMCAUserIdLength;       /* Length of long MCA user */
MQLONG    LongMCAUserIdLength;       /* identifier */
MQLONG    LongRemoteUserIdLength;    /* Length of long remote user */
MQLONG    LongRemoteUserIdLength;    /* identifier */
MQPTR     LongMCAUserIdPtr;          /* Address of long MCA user */
MQPTR     LongMCAUserIdPtr;          /* identifier */
MQPTR     LongRemoteUserIdPtr;       /* Address of long remote */
MQPTR     LongRemoteUserIdPtr;       /* user identifier */
MQBYTE40  MCASecurityId;             /* MCA security identifier */
MQBYTE40  RemoteSecurityId;          /* Remote security identifier */

/* Ver:6 */
MQCHAR    SSLCipherSpec[32];         /* SSL CipherSpec */
MQPTR     SSLPeerNamePtr;            /* Address of SSL peer name */
MQLONG    SSLPeerNameLength;         /* Length of SSL peer name */
MQLONG    SSLClientAuth;             /* Whether SSL client */
MQLONG    SSLClientAuth;             /* authentication is required */
MQLONG    KeepAliveInterval;         /* Keepalive interval */

```

```

MQCHAR    LocalAddress[48];          /* Local communications */
/* address */
MQLONG    BatchHeartbeat;            /* Batch heartbeat interval */
/* Ver:7 */
MQLONG    HdrCompList[2];            /* Header data compression */
/* list */
MQLONG    MsgCompList[16];           /* Message data compression */
/* list */
MQLONG    CLWLChannelRank;           /* Channel rank */
MQLONG    CLWLChannelPriority;        /* Channel priority */
MQLONG    CLWLChannelWeight;         /* Channel weight */
MQLONG    ChannelMonitoring;         /* Channel monitoring */
MQLONG    ChannelStatistics;         /* Channel statistics */
/* Ver:8 */
MQLONG    SharingConversations;      /* Limit on sharing */
/* conversations */
MQLONG    PropertyControl;           /* Message property control */
MQLONG    MaxInstances;              /* Limit on SVRCONN channel */
/* instances */
MQLONG    MaxInstancesPerClient;     /* Limit on SVRCONN channel */
/* instances per client */
MQLONG    ClientChannelWeight;       /* Client channel weight */
MQLONG    ConnectionAffinity;        /* Connection affinity */
/* Ver:9 */
MQLONG    BatchDataLimit;            /* Batch data limit */
MQLONG    UseDLQ;                   /* Use Dead Letter Queue */
MQLONG    DefReconnect;              /* Default client reconnect */
/* option */
/* Ver:10 */
MQCHAR64  CertificateLabel;          /* Certificate label */
/* Ver:11 */
};

```

COBOL declaration

This declaration is the COBOL declaration for the MQCD structure.

```

** MQCD structure
10 MQCD.
  ** Channel definition name
  15 MQCD-CHANNELNAME PIC X(20).
  ** Structure version number
  15 MQCD-VERSION PIC S9(9) BINARY.
  ** Channel type
  15 MQCD-CHANNELTYPE PIC S9(9) BINARY.
  ** Transport type
  15 MQCD-TRANSPORTTYPE PIC S9(9) BINARY.
  ** Channel description
  15 MQCD-DESC PIC X(64).
  ** Queue-manager name
  15 MQCD-QMGRNAME PIC X(48).
  ** Transmission queue name
  15 MQCD-XMITQNAME PIC X(48).
  ** First 20 bytes of connection name
  15 MQCD-SHORTCONNECTIONNAME PIC X(20).
  ** Reserved
  15 MQCD-MCANAME PIC X(20).
  ** LU 6.2 Mode name
  15 MQCD-MODENAME PIC X(8).
  ** LU 6.2 transaction program name
  15 MQCD-TPNAME PIC X(64).
  ** Batch size
  15 MQCD-BATCHSIZE PIC S9(9) BINARY.
  ** Disconnect interval
  15 MQCD-DISCINTERVAL PIC S9(9) BINARY.
  ** Short retry count
  15 MQCD-SHORTRETRYCOUNT PIC S9(9) BINARY.
  ** Short retry wait interval
  15 MQCD-SHORTRETRYINTERVAL PIC S9(9) BINARY.
  ** Long retry count
  15 MQCD-LONGRETRYCOUNT PIC S9(9) BINARY.
  ** Long retry wait interval
  15 MQCD-LONGRETRYINTERVAL PIC S9(9) BINARY.
  ** Channel security exit name
  15 MQCD-SECURITYEXIT PIC X(20).
  ** Channel message exit name
  15 MQCD-MSGEXIT PIC X(20).
  ** Channel send exit name
  15 MQCD-SENDEXIT PIC X(20).
  ** Channel receive exit name

```

```

15 MQCD-RECEIVEEXIT PIC X(20).
** Highest allowable message sequence number
15 MQCD-SEQNUMBERWRAP PIC S9(9) BINARY.
** Maximum message length
15 MQCD-MAXMSGLENGTH PIC S9(9) BINARY.
** Put authority
15 MQCD-PUTAUTHORITY PIC S9(9) BINARY.
** Data conversion
15 MQCD-DATACONVERSION PIC S9(9) BINARY.
** Channel security exit user data
15 MQCD-SECURITYUSERDATA PIC X(32).
** Channel message exit user data
15 MQCD-MSGUSERDATA PIC X(32).
** Channel send exit user data
15 MQCD-SENDUSERDATA PIC X(32).
** Channel receive exit user data
15 MQCD-RECEIVEUSERDATA PIC X(32).
** Ver:1 **
** User identifier
15 MQCD-USERIDENTIFIER PIC X(12).
** Password
15 MQCD-PASSWORD PIC X(12).
** First 12 bytes of MCA user identifier
15 MQCD-MCAUSERIDENTIFIER PIC X(12).
** Message channel agent type
15 MQCD-MCATYPE PIC S9(9) BINARY.
** Connection name
15 MQCD-CONNECTIONNAME PIC X(264).
** First 12 bytes of user identifier from partner
15 MQCD-REMOTEUSERIDENTIFIER PIC X(12).
** Password from partner
15 MQCD-REMOTEPASSWORD PIC X(12).
** Ver:2 **
** Channel message retry exit name
15 MQCD-MSGRETRYEXIT PIC X(20).
** Channel message retry exit user data
15 MQCD-MSGRETRYUSERDATA PIC X(32).
** Number of times MCA will try to put the message, after first
** attempt has failed
15 MQCD-MSGRETRYCOUNT PIC S9(9) BINARY.
** Minimum interval in milliseconds after which the open or put
** operation will be retried
15 MQCD-MSGRETRYINTERVAL PIC S9(9) BINARY.
** Ver:3 **
** Time in seconds between heartbeat flows
15 MQCD-HEARTBEATINTERVAL PIC S9(9) BINARY.
** Batch duration
15 MQCD-BATCHINTERVAL PIC S9(9) BINARY.
** Speed at which nonpersistent messages are sent
15 MQCD-NONPERSISTENTMSGSPPEED PIC S9(9) BINARY.
** Length of MQCD structure
15 MQCD-STRUCLength PIC S9(9) BINARY.
** Length of exit name
15 MQCD-EXITNAMELENGTH PIC S9(9) BINARY.
** Length of exit user data
15 MQCD-EXITDATALENGTH PIC S9(9) BINARY.
** Number of message exits defined
15 MQCD-MSGEXITSDEFINED PIC S9(9) BINARY.
** Number of send exits defined
15 MQCD-SENDEXITSDEFINED PIC S9(9) BINARY.
** Number of receive exits defined
15 MQCD-RECEIVEEXITSDEFINED PIC S9(9) BINARY.
** Address of first MsgExit field
15 MQCD-MSGEXITPTR POINTER.
** Address of first MsgUserData field
15 MQCD-MSGUSERDATAPTR POINTER.
** Address of first SendExit field
15 MQCD-SENDEXITPTR POINTER.
** Address of first SendUserData field
15 MQCD-SENDUSERDATAPTR POINTER.
** Address of first ReceiveExit field
15 MQCD-RECEIVEEXITPTR POINTER.
** Address of first ReceiveUserData field
15 MQCD-RECEIVEUSERDATAPTR POINTER.
** Ver:4 **
** Address of a list of cluster names
15 MQCD-CLUSTERPTR POINTER.
** Number of clusters to which the channel belongs
15 MQCD-CLUSTERSDEFINED PIC S9(9) BINARY.
** Network priority
15 MQCD-NETWORKPRIORITY PIC S9(9) BINARY.
** Ver:5 **

```

```

** Length of long MCA user identifier
15 MQCD-LONGMCAUSERIDLENGTH PIC S9(9) BINARY.
** Length of long remote user identifier
15 MQCD-LONGREMOTEUSERIDLENGTH PIC S9(9) BINARY.
** Address of long MCA user identifier
15 MQCD-LONGMCAUSERIDPTR POINTER.
** Address of long remote user identifier
15 MQCD-LONGREMOTEUSERIDPTR POINTER.
** MCA security identifier
15 MQCD-MCASECURITYID PIC X(40).
** Remote security identifier
15 MQCD-REMOTESECURITYID PIC X(40).
** Ver:6 **
** SSL CipherSpec
15 MQCD-SSLCIPHERSPEC PIC X(32).
** Address of SSL peer name
15 MQCD-SSLPEERNAMEPTR POINTER.
** Length of SSL peer name
15 MQCD-SSLPEERNAMELENGTH PIC S9(9) BINARY.
** Whether SSL client authentication is required
15 MQCD-SSLCLIENTAUTH PIC S9(9) BINARY.
** Keepalive interval
15 MQCD-KEEPALIVEINTERVAL PIC S9(9) BINARY.
** Local communications address
15 MQCD-LOCALADDRESS PIC X(48).
** Batch heartbeat interval
15 MQCD-BATCHHEARTBEAT PIC S9(9) BINARY.
** Ver:7 **
** Header data compression list
15 MQCD-HDRCOMPLIST PIC S9(9) BINARY.
** Message data compression list
15 MQCD-MSGCOMPLIST PIC S9(9) BINARY.
** Channel rank
15 MQCD-CLWLCHANNELRANK PIC S9(9) BINARY.
** Channel priority
15 MQCD-CLWLCHANNELPRIORITY PIC S9(9) BINARY.
** Channel weight
15 MQCD-CLWLCHANNELWEIGHT PIC S9(9) BINARY.
** Channel monitoring
15 MQCD-CHANNELMONITORING PIC S9(9) BINARY.
** Channel statistics
15 MQCD-CHANNELSTATISTICS PIC S9(9) BINARY.
** Ver:8 **
** Limit on sharing conversations
15 MQCD-SHARINGCONVERSATIONS PIC S9(9) BINARY.
** Message property control
15 MQCD-PROPERTYCONTROL PIC S9(9) BINARY.
** Limit on SVRCONN channel instances
15 MQCD-MAXINSTANCES PIC S9(9) BINARY.
** Limit on SVRCONN channel instances per client
15 MQCD-MAXINSTANCESPERCLIENT PIC S9(9) BINARY.
** Client channel weight
15 MQCD-CLIENTCHANNELWEIGHT PIC S9(9) BINARY.
** Connection affinity
15 MQCD-CONNECTIONAFFINITY PIC S9(9) BINARY.
** Ver:9 **
** Batch data limit
15 MQCD-BATCHDATA LIMIT PIC S9(9) BINARY.
** Use Dead Letter Queue
15 MQCD-USEDLQ PIC S9(9) BINARY.
** Default client reconnect option
15 MQCD-DEFRECONNECT PIC S9(9) BINARY.
** Ver:10 **

```

RPG declaration (ILE)

This declaration is the RPG declaration for the MQCD structure.

```

D* MQCD Structure
D*
D* Channel definition name
D CDCHN          1      20
D* Structure version number
D CDVER          21     24I 0
D* Channel type
D CDCHT          25     28I 0
D* Transport type
D CDTRT          29     32I 0
D* Channel description
D CDEDES         33     96

```

```

D* Queue-manager name
D CDQM          97      144
D* Transmission queue name
D CDXQ          145      192
D* First 20 bytes of connection name
D CDSCN         193      212
D* Reserved
D CDMCA         213      232
D* LU 6.2 Mode name
D CDMOD         233      240
D* LU 6.2 transaction program name
D CDTP          241      304
D* Batch size
D CDBS          305      308I 0
D* Disconnect interval
D CDDI          309      312I 0
D* Short retry count
D CDSRC         313      316I 0
D* Short retry wait interval
D CDSRI         317      320I 0
D* Long retry count
D CDLRC         321      324I 0
D* Long retry wait interval
D CDLRI         325      328I 0
D* Channel security exit name
D CDSCX         329      348
D* Channel message exit name
D CDMSX         349      368
D* Channel send exit name
D CDSNX         369      388
D* Channel receive exit name
D CDRCX         389      408
D* Highest allowable message sequence number
D CDSNW         409      412I 0
D* Maximum message length
D CDMML         413      416I 0
D* Put authority
D CDPA          417      420I 0
D* Data conversion
D CDDC          421      424I 0
D* Channel security exit user data
D CDSCD         425      456
D* Channel message exit user data
D CDMSD         457      488
D* Channel send exit user data
D CDSND         489      520
D* Channel receive exit user data
D CDRCD         521      552
D* Ver:1 **
D* User identifier
D CDUID         553      564
D* Password
D CDPW          565      576
D* First 12 bytes of MCA user identifier
D CDAUI         577      588
D* Message channel agent type
D CDCAT         589      592I 0
D* Connection name
D CDCON         593      848
D CDCN2         849      856
D* First 12 bytes of user identifier from partner
D CDRUI         857      868
D* Password from partner
D CDRPW         869      880
D* Ver:2 **
D* Channel message retry exit name
D CDMRX         881      900
D* Channel message retry exit user data
D CDMRD         901      932
D* Number of times MCA will try to put the message, after first
D* attempt has failed
D CDMRC         933      936I 0
D* Minimum interval in milliseconds after which the open or put
D* operation will be retried
D CDMRI         937      940I 0
D* Ver:3 **
D* Time in seconds between heartbeat flows
D CDHBI         941      944I 0
D* Batch duration
D CDBI          945      948I 0
D* Speed at which nonpersistent messages are sent
D CDNPM         949      952I 0

```

```

D* Length of MQCD structure
D CDLEN          953      956I 0
D* Length of exit name
D CDXNL          957      960I 0
D* Length of exit user data
D CDXDL          961      964I 0
D* Number of message exits defined
D CDMXD          965      968I 0
D* Number of send exits defined
D CDSXD          969      972I 0
D* Number of receive exits defined
D CDRXD          973      976I 0
D* Address of first MsgExit field
D CDMXP          977      992*
D* Address of first MsgUserData field
D CDMUP          993     1008*
D* Address of first SendExit field
D CDSXP         1009     1024*
D* Address of first SendUserData field
D CDSUP         1025     1040*
D* Address of first ReceiveExit field
D CDRXP         1041     1056*
D* Address of first ReceiveUserData field
D CDRUP         1057     1072*
D* Ver:4 **
D* Address of a list of cluster names
D CDCLP         1073     1088*
D* Number of clusters to which the channel belongs
D CDCLD         1089     1092I 0
D* Network priority
D CDPN          1093     1096I 0
D* Ver:5 **
D* Length of long MCA user identifier
D CDMLL         1097     1100I 0
D* Length of long remote user identifier
D CDRL          1101     1104I 0
D* Address of long MCA user identifier
D CDLMP         1105     1120*
D* Address of long remote user identifier
D CDLRP         1121     1136*
D* MCA security identifier
D CDMSI         1137     1176
D* Remote security identifier
D CDRSI         1177     1216
D* Ver:6 **
D* SSL CipherSpec
D CDSCS         1217     1248
D* Address of SSL peer name
D CDSPN         1249     1264*
D* Length of SSL peer name
D CDSPL         1265     1268I 0
D* Whether SSL client authentication is required
D CDSCA         1269     1272I 0
D* Keepalive interval
D CDKAI         1273     1276I 0
D* Local communications address
D CDLOA         1277     1324
D* Batch heartbeat interval
D CDBHB         1325     1328I 0
D* Ver:7 **
D* Header data compression list
D CDHCL0
D CDHCL1          1329     1332I 0
D CDHCL2          1333     1336I 0
D CDHCL          10I 0 DIM(2) OVERLAY(CDHCL0)
D* Message data compression list
D CDMCL0
D CDMCL1          1337     1340I 0
D CDMCL2          1341     1344I 0
D CDMCL3          1345     1348I 0
D CDMCL4          1349     1352I 0
D CDMCL5          1353     1356I 0
D CDMCL6          1357     1360I 0
D CDMCL7          1361     1364I 0
D CDMCL8          1365     1368I 0
D CDMCL9          1369     1372I 0
D CDMCL10         1373     1376I 0
D CDMCL11         1377     1380I 0
D CDMCL12         1381     1384I 0
D CDMCL13         1385     1388I 0
D CDMCL14         1389     1392I 0
D CDMCL15         1393     1396I 0

```

D	CDMCL16	1397	1400I	0
D	CDMCL		10I	0 DIM(16) OVERLAY(CDMCL0)
D*	Channel rank			
D	CDCWCR	1401	1404I	0
D*	Channel priority			
D	CDCWCP	1405	1408I	0
D*	Channel weight			
D	CDCWCW	1409	1412I	0
D*	Channel monitoring			
D	CDCHLMON	1413	1416I	0
D*	Channel statistics			
D	CDCHLST	1417	1420I	0
D*	Ver:8 **			
D*	Limit on sharing conversations			
D	CDSHC	1421	1424I	0
D*	Message property control			
D	CDPRC	1425	1428I	0
D*	Limit on SVRCONN channel instances			
D	CDMXIN	1429	1432I	0
D*	Limit on SVRCONN channel instances per client			
D	CDMXIC	1433	1436I	0
D*	Client channel weight			
D	CDCLNCHLW	1437	1440I	0
D*	Connection affinity			
D	CDCONNAFF	1441	1444I	0
D*	Ver:9 **			
D*	Batch data limit			
D	CDBDL	1445	1448I	0
D*	Use Dead Letter Queue			
D	CDUDLQ	1449	1452I	0
D*	Default client reconnect option			
D	CDDRCN	1453	1456I	0
D*	Ver:10 **			

System/390 assembler declaration

This declaration is the System/390 assembler declaration for the MQCD structure.

MQCD	DSECT		
MQCD_CHANNELNAME	DS	CL20	Channel definition name
MQCD_VERSION	DS	F	Structure version number
MQCD_CHANNELTYPE	DS	F	Channel type
MQCD_TRANSPORTTYPE	DS	F	Transport type
MQCD_DESC	DS	CL64	Channel description
MQCD_QMGRNAME	DS	CL48	Queue-manager name
MQCD_XMITQNAME	DS	CL48	Transmission queue name
MQCD_SHORTCONNECTIONNAME	DS	CL20	First 20 bytes of connection name
*			
MQCD_MCANAME	DS	CL20	Reserved
MQCD_MODENAME	DS	CL8	LU 6.2 Mode name
MQCD_TPNAME	DS	CL64	LU 6.2 transaction program name
MQCD_BATCHSIZE	DS	F	Batch size
MQCD_DISCINTERVAL	DS	F	Disconnect interval
MQCD_SHORTRETRYCOUNT	DS	F	Short retry count
MQCD_SHORTRETRYINTERVAL	DS	F	Short retry wait interval
MQCD_LONGRETRYCOUNT	DS	F	Long retry count
MQCD_LONGRETRYINTERVAL	DS	F	Long retry wait interval
MQCD_SECURITYEXIT	DS	CLn	Channel security exit name
MQCD_MSGEXIT	DS	CLn	Channel message exit name
MQCD_SENDEXIT	DS	CLn	Channel send exit name
MQCD_RECEIVEEXIT	DS	CLn	Channel receive exit name
MQCD_SEQNUMBERWRAP	DS	F	Highest allowable message sequence number
*			
MQCD_MAXMSGLLENGTH	DS	F	Maximum message length
MQCD_PUTAUTHORITY	DS	F	Put authority
MQCD_DATACONVERSION	DS	F	Data conversion
MQCD_SECURITYUSERDATA	DS	CL32	Channel security exit user data
MQCD_MSGUSERDATA	DS	CL32	Channel message exit user data
MQCD_SENDUSERDATA	DS	CL32	Channel send exit user data
MQCD_RECEIVEUSERDATA	DS	CL32	Channel receive exit user data
MQCD_USERIDENTIFIER	DS	CL12	User identifier
MQCD_PASSWORD	DS	CL12	Password
MQCD_MCAUSERIDENTIFIER	DS	CL12	First 12 bytes of MCA user identifier
*			
MQCD_MCATYPE	DS	F	Message channel agent type
MQCD_CONNECTIONNAME	DS	CL264	Connection name
MQCD_REMOTEUSERIDENTIFIER	DS	CL12	First 12 bytes of user identifier from partner
*			
MQCD_REMOTEPASSWORD	DS	CL12	Password from partner
MQCD_MSGRETRYEXIT	DS	CLn	Channel message retry exit name

MQCD_MSGRETRYUSERDATA	DS	CL32	Channel message retry exit user data
* MQCD_MSGRETRYCOUNT	DS	F	Number of times MCA will try to put the message, after the first attempt has failed
* MQCD_MSGRETRYINTERVAL	DS	F	Minimum interval in milliseconds after which the open or put operation will be retried
* MQCD_HEARTBEATINTERVAL	DS	F	Time in seconds between heartbeat flows
* MQCD_BATCHINTERVAL	DS	F	Batch duration
* MQCD_NONPERSISTENTMSGSPPEED	DS	F	Speed at which nonpersistent messages are sent
* MQCD_STRUCLNGTH	DS	F	Length of MQCD structure
* MQCD_EXITNAMELENGTH	DS	F	Length of exit name
* MQCD_EXITDATALENGTH	DS	F	Length of exit user data
* MQCD_MSGEXITSDEFINED	DS	F	Number of message exits defined
* MQCD_SENDEXITSDEFINED	DS	F	Number of send exits defined
* MQCD_RECEIVEEXITSDEFINED	DS	F	Number of receive exits defined
* MQCD_MSGEXITPTR	DS	F	Address of first MSGEXIT field
* MQCD_MSGUSERDATAPTR	DS	F	Address of first MSGUSERDATA field
* MQCD_SENDEXITPTR	DS	F	Address of first SENDEXIT field
* MQCD_SENDUSERDATAPTR	DS	F	Address of first SENDUSERDATA field
* MQCD_RECEIVEEXITPTR	DS	F	Address of first RECEIVEEXIT field
* MQCD_RECEIVEUSERDATAPTR	DS	F	Address of first RECEIVEUSERDATA field
* MQCD_CLUSTERPTR	DS	F	Address of a list of cluster names
* MQCD_CLUSTERSDEFINED	DS	F	Number of clusters to which the channel belongs
* MQCD_NETWORKPRIORITY	DS	F	Network priority
* MQCD_LONGMCAUSERIDLENGTH	DS	F	Length of long MCA user identifier
* MQCD_LONGREMOTEUSERIDLENGTH	DS	F	Length of long remote user identifier
* MQCD_LONGMCAUSERIDPTR	DS	F	Address of long MCA user identifier
* MQCD_LONGREMOTEUSERIDPTR	DS	F	Address of long remote user identifier
* MQCD_MCASESECURITYID	DS	XL40	MCA security identifier
* MQCD_REMOTESECURITYID	DS	XL40	Remote security identifier
* MQCD_SSLCIPHERSPEC	DS	CL32	SSL CipherSpec
* MQCD_SSLPEERNAMEPTR	DS	F	Address of SSL peer name
* MQCD_SSLPEERNAMELENGTH	DS	F	Length of SSL peer name
* MQCD_SSLCLIENTAUTH	DS	F	Whether SSL client authentication is required
* MQCD_KEEPLIVEINTERVAL	DS	F	Keepalive interval
* MQCD_LOCALADDRESS	DS	CL48	Local communications address
* MQCD_BATCHHEARTBEAT	DS	F	Batch heartbeat interval
* MQCD_HDRCOMPLIST	DS	CL2	Header data compression list
* MQCD_MSGCOMPLIST	DS	CL16	Message data compression list
* MQCD_CLWLCHANNELRANK	DS	F	Channel rank
* MQCD_CLWLCHANNELPRIORITY	DS	F	Channel priority
* MQCD_CLWLCHANNELWEIGHT	DS	F	Channel weight
* MQCD_CHANNELMONITORING	DS	F	Channel monitoring
* MQCD_CHANNELSTATISTICS	DS	F	Channel statistics
* MQCD_SHARINGCONVERSATIONS	DS	F	Limit on sharing conversations
* MQCD_PROPERTYCONTROL	DS	F	Message property control
* MQCD_SHARINGCONVERSATIONS	DS	F	Limit on sharing conversations
* MQCD_PROPERTYCONTROL	DS	F	Message property control
* MQCD_MAXINSTANCES	DS	F	Limit on SVRCONN chl instances
* MQCD_MAXINSTANCESPERCLIENT	DS	F	Limit on SVRCONN chl instances per client
* MQCD_CLIENTCHANNELWEIGHT	DS	F	Channel weight
* MQCD_CONNECTIONAFFINITY	DS	F	Connection Affinty
* MQCD_BATCHDATA LIMIT	DS	F	Batch data limit
* MQCD_USED LQ	DS	F	Use dead-letter queue
* MQCD_DEFRECONNECT	DS	F	Default client reconnect option
* MQCD_LENGTH	EQU	*-MQCD	
* MQCD_AREA	ORG	MQCD	
	DS	CL(MQCD_LENGTH)	

Visual Basic declaration

This declaration is the Visual Basic declaration of the MQCD structure.

In Visual Basic, the MQCD structure can be used with the MQCNO structure on the MQCONN call.

```

Type MQCD
  ChannelName      As String*20 'Channel definition name'
  Version          As Long      'Structure version number'
  ChannelType      As Long      'Channel type'
  TransportType    As Long      'Transport type'
  Desc             As String*64 'Channel description'
  QMgrName         As String*48 'Queue-manager name'
  XmitQName        As String*48 'Transmission queue name'
  ShortConnectionName As String*20 'First 20 bytes of connection'
                                     'name'
  MCAName          As String*20 'Reserved'
  ModeName         As String*8  'LU 6.2 Mode name'
  TpName           As String*64 'LU 6.2 transaction program name'
  BatchSize        As Long      'Batch size'
  DiscInterval     As Long      'Disconnect interval'
  ShortRetryCount  As Long      'Short retry count'
  ShortRetryInterval As Long    'Short retry wait interval'
  LongRetryCount   As Long      'Long retry count'
  LongRetryInterval As Long    'Long retry wait interval'
  SecurityExit     As String*128 'Channel security exit name'
  MsgExit          As String*128 'Channel message exit name'
  SendExit         As String*128 'Channel send exit name'
  ReceiveExit      As String*128 'Channel receive exit name'
  SeqNumberWrap    As Long      'Highest allowable message'
                                     'sequence number'
  MaxMsgLength     As Long      'Maximum message length'
  PutAuthority     As Long      'Put authority'
  DataConversion   As Long      'Data conversion'
  SecurityUserData As String*32 'Channel security exit user data'
  MsgUserData      As String*32 'Channel message exit user data'
  SendUserData     As String*32 'Channel send exit user data'
  ReceiveUserData  As String*32 'Channel receive exit user data'
  UserIdentifier   As String*12 'User identifier'
  Password         As String*12 'Password'
  MCAUserIdentifier As String*12 'First 12 bytes of MCA user'
                                     'identifier'
  MCAType          As Long      'Message channel agent type'
  ConnectionName   As String*264 'Connection name'
  RemoteUserIdentifier As String*12 'First 12 bytes of user'
                                     'identifier from partner'
  RemotePassword   As String*12 'Password from partner'
  MsgRetryExit     As String*128 'Channel message retry exit name'
  MsgRetryUserData As String*32 'Channel message retry exit user'
                                     'data'
  MsgRetryCount    As Long      'Number of times MCA will try to'
                                     'put the message, after the'
                                     'first attempt has failed'
  MsgRetryInterval As Long      'Minimum interval in'
                                     'milliseconds after which the'
                                     'open or put operation will be'
                                     'retried'
  HeartbeatInterval As Long      'Time in seconds between'
                                     'heartbeat flows'
  BatchInterval     As Long      'Batch duration'
  NonPersistentMsgSpeed As Long  'Speed at which nonpersistent'
                                     'messages are sent'
  StrucLength       As Long      'Length of MQCD structure'
  ExitNameLength    As Long      'Length of exit name'
  ExitDataLength    As Long      'Length of exit user data'
  MsgExitsDefined   As Long      'Number of message exits defined'
  SendExitsDefined  As Long      'Number of send exits defined'
  ReceiveExitsDefined As Long    'Number of receive exits defined'
  MsgExitPtr        As MQPTR     'Address of first MsgExit field'
  MsgUserDataPtr    As MQPTR     'Address of first MsgUserData'
                                     'field'
  SendExitPtr       As MQPTR     'Address of first SendExit field'
  SendUserDataPtr   As MQPTR     'Address of first SendUserData'
                                     'field'
  ReceiveExitPtr    As MQPTR     'Address of first ReceiveExit'
                                     'field'
  ReceiveUserDataPtr As MQPTR     'Address of first'
                                     'ReceiveUserData field'
  ClusterPtr        As MQPTR     'Address of a list of cluster'
                                     'names'
  ClustersDefined   As Long      'Number of clusters to which the'

```

NetworkPriority	As Long	'channel belongs'
LongMCAUserIdLength	As Long	'Network priority'
		'Length of long MCA user'
		'identifier'
LongRemoteUserIdLength	As Long	'Length of long remote user'
		'identifier'
LongMCAUserIdPtr	As MQPTR	'Address of long MCA user'
		'identifier'
LongRemoteUserIdPtr	As MQPTR	'Address of long remote user'
		'identifier'
MCASecurityId	As MQBYTE40	'MCA security identifier'
RemoteSecurityId	As MQBYTE40	'Remote security identifier'
SSLCipherSpec	As String*32	'SSL CipherSpec'
SSLPeerNamePtr	As MQPTR	'Address of SSL peer name'
SSLPeerNameLength	As Long	'Length of SSL peer name'
SSLClientAuth	As Long	'Whether SSL client'
		'authentication is required'
KeepAliveInterval	As Long	'Keepalive interval'
LocalAddress	As String*48	'Local communications address'
BatchHeartbeat	As Long	'Batch heartbeat interval'
HdrCompList(0 to 1)	As Long2	'Header data compression list'
MsgCompList(0 To 15)	As Long16	'Message data compression list'
CLWLChannelRank	As Long	'Channel Rank'
CLWLChannelPriority	As Long	'Channel priority'
CLWLChannelWeight	As Long	'Channel Weight'
ChannelMonitoring	As Long	'Channel Monitoring control'
ChannelStatistics	As Long	'Channel Statistics'
End Type		

Changing MQCD fields in a channel exit

A channel exit can change fields in the MQCD. However, these changes are not typically acted on, except in the circumstances listed.

If a channel exit program changes a field in the MQCD data structure, the new value is typically ignored by the IBM MQ channel process. However, the new value remains in the MQCD and is passed to any remaining exits in an exit chain and to any conversation sharing the channel instance.

If SharingConversations is set to FALSE in the MQCXP structure, changes to certain fields can be acted on, depending on the type of exit program, the type of channel, and the exit reason code. The following table shows the fields that can be changed and affect the behavior of the channel, and in what circumstances. If an exit program changes one of these fields in any other circumstances, or any field not listed, the new value is ignored by the channel process. The new value remains in the MQCD and is passed to any remaining exits in an exit chain and to any conversation sharing the channel instance.

Any type of exit program when called for initialization (MQXR_INIT) can change the ChannelName field of any type of channel, as long as MQCXP SharingConversations is set to FALSE. Only a security exit can change the MCAUserIdentifier field, regardless of the value of MQCXP SharingConversations.

Field	Exit Reason Code	Exit Type	Channel Type
ChannelName	MQXR_INIT	All	All
TransportType	MQXR_INIT	All	All
XmitQName	MQXR_INIT	All	SDR, RCVR
ModeName	MQXR_INIT	All	All
TpName	MQXR_INIT	All	All
BatchSize	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR

Field	Exit Reason Code	Exit Type	Channel Type
DiscInterval	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
ShortRetryCount	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
ShortRetryInterval	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
LongRetryCount	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
LongRetryInterval	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
SeqNumberWrap	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
MaxMsgLength	MQXR_INIT	All	All
PutAuthority	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
DataConversion	MQXR_INIT	All	All
MCAUserIdentifier	MQXR_INIT, MQXR_INIT_SEC, MQXR_SEC_MSG, MQXR_SEC_PARMS	Security	RCVR, RQSTR, SVRCONN, CLUSRCVR
ConnectionName	MQXR_INIT	All	SDR, SVR, RQSTR, CLNTCONN, CLUSSDR, CLUSRCVR

Field	Exit Reason Code	Exit Type	Channel Type
MsgRetryUserData	MQXR_INIT	All	RCVR, RQSTR, CLUSRCVR
MsgRetryCount	MQXR_INIT	All	RCVR, RQSTR, CLUSRCVR
MsgRetryInterval	MQXR_INIT	All	RCVR, RQSTR, CLUSRCVR
HeartbeatInterval	MQXR_INIT	All	All
BatchInterval	MQXR_INIT	All	SDR, SVR, CLUSSDR, CLUSRCVR
NonPersistentMsgSpeed	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
MCASecurityId	MQXR_INIT, MQXR_INIT_SEC, MQXR_SEC_MSG, MQXR_SEC_PARMS	Security	SDR, SVR, RCVR, RQSTR, SVRCONN, CLUSSDR, CLUSRCVR
SSLCipherSpec	MQXR_INIT	All	All
SSLPeerNamePtr	MQXR_INIT	All	All
SSLPeerNameLength	MQXR_INIT	All	All
SSLClientAuth	MQXR_INIT	All	SVR, RCVR, RQSTR, SVRCONN, CLUSRCVR
KeepAliveInterval	MQXR_INIT	All	All
LocalAddress	MQXR_INIT	All	SDR, SVR, RQSTR, CLNTCONN, CLUSSDR, CLUSRCVR
BatchHeartbeat	MQXR_INIT	All	SDR, SVR, CLUSSDR, CLUSRCVR
HdrCompList	MQXR_INIT	All	All
MsgCompList	MQXR_INIT	All	All

Field	Exit Reason Code	Exit Type	Channel Type
ChannelMonitoring	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, SVRCONN, CLUSSDR, CLUSRCVR
ChannelStatistics	MQXR_INIT	All	SDR, SVR, RCVR, RQSTR, CLUSSDR, CLUSRCVR
SharingConversations	MQXR_INIT	All	SVRCONN, CLNTCONN
PropertyControl	MQXR_INIT	All	SDR, SVR, CLUSSDR, CLUSRCVR

MQCXP - Channel exit parameter

The MQCXP structure is passed to each type of exit called by a Message Channel Agent (MCA), client-connection channel, or server-connection channel.

See MQ_CHANNEL_EXIT.

The fields described as "input to the exit" in the descriptions that follow are ignored by the channel when the exit returns control to the channel. Any input fields that the exit changes in the channel exit parameter block will not be preserved for its next invocation. Changes made to input/output fields (for example, the *ExitUserArea* field), are preserved for invocations of that instance of the exit only. Such changes cannot be used to pass data between different exits defined on the same channel, or between the same exit defined on different channels.

Related reference

[“Fields” on page 1526](#)

This topic lists all the fields in the MQCXP structure and describes each field.

[“C declaration” on page 1537](#)

This declaration is the C declaration for the MQCXP structure.

[“COBOL declaration” on page 1538](#)

This declaration is the COBOL declaration for the MQCXP structure.

[“RPG declaration \(ILE\)” on page 1539](#)

This declaration is the RPG declaration for the MQCXP structure.

[“System/390 assembler declaration” on page 1539](#)

This declaration is the System/390 assembler declaration for the MQCXP structure.

Fields

This topic lists all the fields in the MQCXP structure and describes each field.

StrucId (MQCHAR4)

This field specifies the structure identifier.

The value must be:

MQCXP_STRUC_ID

Identifier for channel exit parameter structure.

For the C programming language, the constant MQCXP_STRUC_ID_ARRAY is also defined; this constant has the same value as MQCXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)

This field specifies the structure version number.

The value depends on the environment:

MQCXP_VERSION_1

Version-1 channel exit parameter structure.

MQCXP_VERSION_2

Version-2 channel exit parameter structure.

The field has this value in the HP Integrity NonStop Server environment.

MQCXP_VERSION_3

Version-3 channel exit parameter structure.

The field has this value in the following environments: UNIX systems not listed elsewhere.

MQCXP_VERSION_4

Version-4 channel exit parameter structure.

MQCXP_VERSION_5

Version-5 channel exit parameter structure.

MQCXP_VERSION_6

Version-6 channel exit parameter structure.

MQCXP_VERSION_8

Version-8 channel exit parameter structure.

The field has this value in the following environments: z/OS, AIX, HP-UX, Linux, IBM i, Solaris, Windows.

Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions of the fields. The following constant specifies the version number of the current version:

MQCXP_CURRENT_VERSION

Current version of channel exit parameter structure.

The value depends on the environment.

Note: When a new version of the MQCXP structure is introduced, the layout of the existing part is not changed. The exit must therefore check that the version number is equal to or greater than the lowest version which contains the fields that the exit needs to use.

This is an input field to the exit.

ExitId (MQLONG)

This field specifies the type of exit being called and is set on entry to the exit routine.

The following values are possible:

MQXT_CHANNEL_SEC_EXIT

Channel security exit.

MQXT_CHANNEL_MSG_EXIT

Channel message exit.

MQXT_CHANNEL_SEND_EXIT

Channel send exit.

MQXT_CHANNEL_RCV_EXIT

Channel receive exit.

MQXT_CHANNEL_MSG_RETRY_EXIT

Channel message-retry exit.

MQXT_CHANNEL_AUTO_DEF_EXIT

Channel auto-definition exit.

On z/OS, this type of exit is supported only for channels of type MQCHT_CLUSSDR and MQCHT_CLUSRCVR.

This is an input field to the exit.

ExitReason (MQLONG)

This field specifies the reason why the exit is being called and is set on entry to the exit routine.

It is not used by the auto-definition exit. The following values are possible:

MQXR_INIT

Exit initialization.

This value indicates that the exit is being invoked for the first time. It allows the exit to acquire and initialize any resources that it needs (for example: memory).

MQXR_TERM

Exit termination.

This value indicates that the exit is about to be terminated. The exit should free any resources that it has acquired since it was initialized (for example: memory).

MQXR_MSG

Process a message.

This value indicates that the exit is being invoked to process a message. This value occurs for channel message exits only.

MQXR_XMIT

Process a transmission.

This value occurs for channel send and receive exits only.

MQXR_SEC_MSG

Security message received.

This value occurs for channel security exits only.

MQXR_INIT_SEC

Initiate security exchange.

This value occurs for channel security exits only.

The security exit of the receiver is always invoked with this reason immediately after being invoked with MQXR_INIT, to give it the opportunity to initiate a security exchange. If it declines the opportunity (by returning MQXCC_OK instead of MQXCC_SEND_SEC_MSG or MQXCC_SEND_AND_REQUEST_SEC_MSG), the security exit of the sender is invoked with MQXR_INIT_SEC.

If the security exit of the receiver does initiate a security exchange (by returning MQXCC_SEND_SEC_MSG or MQXCC_SEND_AND_REQUEST_SEC_MSG), the security exit of the sender is never invoked with MQXR_INIT_SEC; instead it is invoked with MQXR_SEC_MSG to process the message of the receiver. (In either case it is first invoked with MQXR_INIT.)

Unless one of the security exits requests termination of the channel (by setting *ExitResponse* to MQXCC_SUPPRESS_FUNCTION or MQXCC_CLOSE_CHANNEL), the security exchange must complete at the side that initiated the exchange. Therefore, if a security exit is invoked with MQXR_INIT_SEC and it does initiate an exchange, the next time the exit is invoked it will be with MQXR_SEC_MSG. This happens whether there is a security message for the exit to process or not. There is a security message if the partner returns MQXCC_SEND_SEC_MSG or MQXCC_SEND_AND_REQUEST_SEC_MSG, but not if the partner returns MQXCC_OK or there is no security exit at the partner. If there is no

security message to process, the security exit at the initiating end is re-invoked with a *DataLength* of zero.

MQXR_RETRY

Retry a message.

This value occurs for message-retry exits only.

MQXR_AUTO_CLUSSDR

Automatic definition of a cluster-sender channel.

This value occurs for channel auto-definition exits only.

MQXR_AUTO_RECEIVER

Automatic definition of a receiver channel.

This value occurs for channel auto-definition exits only.

MQXR_AUTO_SVRCONN

Automatic definition of a server-connection channel.

This value occurs for channel auto-definition exits only.

MQXR_AUTO_CLUSRCVR

Automatic definition of a cluster-receiver channel.

This value occurs for channel auto-definition exits only.

MQXR_SEC_PARMS

Security parameters

This value applies to security exits only and indicates that an MQCSP structure is being passed to the exit. For more information, see [“MQCSP - Security parameters” on page 320](#)

Note:

1. If you have more than one exit defined for a channel, they are each invoked with MQXR_INIT when the MCA is initialized. Also, they are each invoked with MQXR_TERM when the MCA is terminated.
2. For the channel auto-definition exit, *ExitReason* is not set if *Version* is less than MQCXP_VERSION_4. The value MQXR_AUTO_SVRCONN is implied in this case.

This is an input field to the exit.

ExitResponse (MQLONG)

This field specifies the response from the exit.

This field is set by the exit to communicate with the MCA. It must be one of the following values:

MQXCC_OK

Exit completed successfully.

- For the channel security exit, this value indicates that message transfer can now proceed normally.
- For the channel message retry exit, this value indicates that the MCA must wait for the time interval returned by the exit in the *MsgRetryInterval* field in MQCXP, and then try the message again.

The *ExitResponse2* field might contain additional information.

MQXCC_SUPPRESS_FUNCTION

Suppress function.

- For the channel security exit, this value indicates that the channel must be terminated.
- For the channel message exit, this value indicates that the message is not to proceed any further towards its destination. Instead the MCA generates an exception report message (if one was requested by the sender of the original message), and places the message contained in the original buffer on the dead-letter queue (if the sender specified MQRO_DEAD_LETTER_Q), or discards it (if the sender specified MQRO_DISCARD_MSG).

For persistent messages, if the sender specified MQRO_DEAD_LETTER_Q, but the put to the dead-letter queue fails, or there is no dead-letter queue, the original message is left on the transmission queue and the report message is not generated. The original message is also left on the transmission queue if the report message cannot be generated successfully.

The *Feedback* field in the MQDLH structure at the start of the message on the dead-letter queue indicates why the message was put on the dead-letter queue; this feedback code is also used in the message descriptor of the exception report message (if one was requested by the sender).

- For the channel message retry exit, this value indicates that the MCA does not wait and try the message again; instead, the MCA continues immediately with its normal failure processing (the message is placed on the dead-letter queue or discarded, as specified by the sender of the message).
- For the channel auto-definition exit, either MQXCC_OK or MQXCC_SUPPRESS_FUNCTION must be specified. If neither of these values is specified, MQXCC_SUPPRESS_FUNCTION is assumed by default and the auto-definition is abandoned.

This response is not supported for the channel send and receive exits.

MQXCC_SEND_SEC_MSG

Send security message.

This value can be set only by a channel security exit. It indicates that the exit has provided a security message which must be transmitted to the partner.

MQXCC_SEND_AND_REQUEST_SEC_MSG

Send security message that requires a reply.

This value can be set only by a channel security exit. It indicates

- that the exit has provided a security message which can be transmitted to the partner, and
- that the exit requires a response from the partner. If no response is received, the channel must be terminated, because the exit has not yet decided whether communications can proceed.

MQXCC_SUPPRESS_EXIT

Suppress exit.

- This value can be set by all types of channel exit other than a security exit or an auto-definition exit. It suppresses any further invocation of that exit (as if its name had been blank in the channel definition), until termination of the channel, when the exit is again invoked with an *ExitReason* of MQXR_TERM.
- If a message retry exit returns this value, message retries for subsequent messages are controlled by the *MsgRetryCount* and *MsgRetryInterval* channel attributes as normal. For the current message, the MCA performs the number of outstanding retries, at intervals given by the *MsgRetryInterval* channel attribute, but only if the reason code is one that the MCA would normally retry (see the *MsgRetryCount* field described in “MQCD - Channel definition” on page 1487). The number of outstanding retries is the value of the *MsgRetryCount* attribute, less the number of times the exit returned MQXCC_OK for the current message; if this number is negative, no further retries are performed by the MCA for the current message.

MQXCC_CLOSE_CHANNEL

Close channel.

This value can be set by any type of channel exit except an auto-definition exit.

If sharing conversations is not enabled, this value closes the channel.

If sharing conversations is enabled, this value ends the conversation. If this conversation is the only conversation on the channel, the channel also closes.

This field is an input/output field from the exit.

ExitResponse2 (MQLONG)

This field specifies the secondary response from the exit.

This field is set to zero on entry to the exit routine. It can be set by the exit to provide further information to the IBM MQ channel functions. It is not used by the auto-definition exit.

The exit can set one or more of the following values. If more than one is required, the values are added. Combinations that are not valid are noted; other combinations are allowed.

MQXR2_PUT_WITH_DEF_ACTION

Put with default action.

This value is set by the channel message exit of the receiver. It indicates that the message is to be put with the default action of the MCA, that is either the default user ID of the MCA, or the context *UserIdentifier* in the MQMD (message descriptor) of the message.

The value is zero, which corresponds to the initial value set when the exit is invoked. The constant is provided for documentation purposes.

MQXR2_PUT_WITH_DEF_USERID

Put with default user identifier.

This value can only be set by the channel message exit of the receiver. It indicates that the message is to be put with the default user identifier of the MCA.

MQXR2_PUT_WITH_MSG_USERID

Put with user identifier of the message.

This value can only be set by the channel message exit of the receiver. It indicates that the message is to be put with the context *UserIdentifier* in the MQMD (message descriptor) of the message (this might have been modified by the exit).

Only one of MQXR2_PUT_WITH_DEF_ACTION, MQXR2_PUT_WITH_DEF_USERID, and MQXR2_PUT_WITH_MSG_USERID should be set.

MQXR2_USE_AGENT_BUFFER

Use agent buffer.

This value indicates that any data to be passed on is in *AgentBuffer*, not *ExitBufferAddr*.

The value is zero, which corresponds to the initial value set when the exit is invoked. The constant is provided for documentation purposes.

MQXR2_USE_EXIT_BUFFER

Use exit buffer.

This value indicates that any data to be passed on is in *ExitBufferAddr*, not *AgentBuffer*.

Only one of MQXR2_USE_AGENT_BUFFER and MQXR2_USE_EXIT_BUFFER should be set.

MQXR2_DEFAULT_CONTINUATION

Default continuation.

Continuation with the next exit in the chain depends on the response from the last exit invoked:

- If MQXCC_SUPPRESS_FUNCTION or MQXCC_CLOSE_CHANNEL are returned, no further exits in the chain are called.
- Otherwise, the next exit in the chain is invoked.

MQXR2_CONTINUE_CHAIN

Continue with the next exit.

MQXR2_SUPPRESS_CHAIN

Skip remaining exits in chain.

This is an input/output field to the exit.

Feedback (MQLONG)

This field specifies the feedback code.

This field is set to MQFB_NONE on entry to the exit routine.

If a channel message exit sets the *ExitResponse* field to MQXCC_SUPPRESS_FUNCTION, the *Feedback* field specifies the feedback code that identifies why the message was put on the dead-letter (undelivered-message) queue, and is also used to send an exception report if one has been requested. In this case, if the *Feedback* field is MQFB_NONE, the following feedback code is used:

MQFB_STOPPED_BY_MSG_EXIT

Message stopped by channel message exit.

The value returned in this field by channel security, send, receive, and message-retry exits is not used by the MCA.

The value returned in this field by auto-definition exits is not used if *ExitResponse* is MQXCC_OK, but otherwise is used for the *AuxErrorDataInt1* parameter in the event message.

This is an input/output field from the exit.

MaxSegmentLength (MQLONG)

This field specifies the maximum length in bytes that can be sent in a single transmission.

It is not used by the auto-definition exit. It is of interest to a channel send exit, because this exit must ensure that it does not increase the size of a transmission segment to a value greater than *MaxSegmentLength*. The length includes the initial 8 bytes that the exit must not change. The value is negotiated between the IBM MQ channel functions when the channel is initiated. See [Writing channel-exit programs](#) for more information about segment lengths.

The value in this field is not meaningful if *ExitReason* is MQXR_INIT.

This is an input field to the exit.

ExitUserArea (MQBYTE16)

This field specifies the exit user area - a field available for the exit to use.

It is initialized to binary zero before the first invocation of the exit (which has an *ExitReason* set to MQXR_INIT), and thereafter any changes made to this field by the exit are preserved across invocations of the exit.

The following value is defined:

MQXUA_NONE

No user information.

The value is binary zero for the length of the field.

For the C programming language, the constant MQXUA_NONE_ARRAY is also defined; this constant has the same value as MQXUA_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH. This is an input/output field to the exit.

ExitData (MQCHAR32)

This field specifies the exit data.

This field is set on entry to the exit routine to information that IBM MQ channel functions took from the channel definition. If no such information is available, this field is all blanks.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

This is an input field to the exit.

The following fields in this structure are not present if *Version* is less than MQCXP_VERSION_2.

MsgRetryCount (MQLONG)

This field specifies the number of times the message has been retried.

The first time the exit is invoked for a particular message, this field has the value zero (no retries yet attempted). On each subsequent invocation of the exit for that message, the value is incremented by one by the MCA.

This is an input field to the exit. The value in this field is not meaningful if *ExitReason* is MQXR_INIT. The field is not present if *Version* is less than MQCXP_VERSION_2.

MsgRetryInterval (MQLONG)

This field specifies the minimum interval in milliseconds after which the put operation is retried.

The first time the exit is invoked for a particular message, this field contains the value of the *MsgRetryInterval* channel attribute. The exit can leave the value unchanged, or modify it to specify a different time interval in milliseconds. If the exit returns MQXCC_OK in *ExitResponse*, the MCA waits for at least this time interval before retrying the MQOPEN or MQPUT operation. The time interval specified must be zero or greater.

The second and subsequent times the exit is invoked for that message, this field contains the value returned by the previous invocation of the exit.

If the value returned in the *MsgRetryInterval* field is less than zero or greater than 999 999 999, and *ExitResponse* is MQXCC_OK, the MCA ignores the *MsgRetryInterval* field in MQCXP and waits instead for the interval specified by the *MsgRetryInterval* channel attribute.

This is an input/output field to the exit. The value in this field is not meaningful if *ExitReason* is MQXR_INIT. The field is not present if *Version* is less than MQCXP_VERSION_2.

MsgRetryReason (MQLONG)

This field specifies the reason code from the previous attempt to put the message.

This field is the reason code from the previous attempt to put the message; it is one of the MQRC_* values.

This is an input field to the exit. The value in this field is not meaningful if *ExitReason* is MQXR_INIT. The field is not present if *Version* is less than MQCXP_VERSION_2.

The following fields in this structure are not present if *Version* is less than MQCXP_VERSION_3.

HeaderLength (MQLONG)

This field specifies the length of header information.

This field is relevant only for a message exit and a message-retry exit. The value is the length of the routing header structures at the start of the message data; these are the MQXQH structure, the MQMDE (message description extension header), and (for a distribution-list message) the MQDH structure and arrays of MQOR and MQPMR records that follow the MQXQH structure.

The message exit can examine this header information, and modify it if necessary, but the data that the exit returns must still be in the correct format. The exit must not, for example, encrypt or compress the header data at the sending end, even if the message exit at the receiving end makes compensating changes.

If the message exit modifies the header information in such a way as to change its length (for example, by adding another destination to a distribution-list message), it must change the value of *HeaderLength* correspondingly before returning.

This is an input/output field to the exit. The value in this field is not meaningful if *ExitReason* is MQXR_INIT. The field is not present if *Version* is less than MQCXP_VERSION_3.

PartnerName (MQCHAR48)

This field specifies the name of the partner.

The name of the partner, as follows:

- For SVRCONN channels, it is the logged-on user ID at the client.

- For all other types of channel, it is the queue-manager name of the partner.

When the exit is initialized this field is blank because the queue manager does not know the name of the partner until after initial negotiation has taken place.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_3.

FAPLevel (MQLONG)

Negotiated Formats and Protocols level.

This is an input field to the exit. Changes to this field should only be made under the direction of IBM service. The field is not present if *Version* is less than MQCXP_VERSION_3.

CapabilityFlags (MQLONG)

This field specifies the capability flags.

The following are defined:

MQCF_NONE

No flags.

MQCF_DIST_LISTS

Distribution lists supported.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_3.

ExitNumber (MQLONG)

This field specifies the ordinal number of the exit.

The ordinal number of the exit, within the type defined in *ExitId*. For example, if the exit being invoked is the third message exit defined, this field contains the value 3. If the exit type is one for which a list of exits cannot be defined (for example, a security exit), this field has the value 1.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_3.

The following fields in this structure are not present if *Version* is less than MQCXP_VERSION_5.

ExitSpace (MQLONG)

This field specifies the number of bytes in the transmission buffer reserved for the exit to use.

This field is relevant only for a send exit. It specifies the amount of space in bytes that the IBM MQ channel functions reserve in the transmission buffer for the exit to use. This field allows the exit to add to the transmission buffer a small amount of data (typically not exceeding a few hundred bytes) for use by a complementary receive exit at the other end. The data added by the send exit must be removed by the receive exit.

The value is always zero on z/OS.

Note: This facility must not be used to send large amounts of data, as it might degrade performance, or even inhibit operation of the channel.

By setting *ExitSpace* the exit is guaranteed that there is always at least that number of bytes available in the transmission buffer for the exit to use. However, the exit can use less than the amount reserved, or more than the amount reserved if there is space available in the transmission buffer. The exit space in the buffer is provided following the existing data.

ExitSpace can be set by the exit only when *ExitReason* has the value MQXR_INIT; in all other cases the value returned by the exit is ignored. On input to the exit, *ExitSpace* is zero for the MQXR_INIT call, and is the value returned by the MQXR_INIT call in other cases.

If the value returned by the MQXR_INIT call is negative, or there are fewer than 1024 bytes available in the transmission buffer for message data after reserving the requested exit space for all the send exits in the chain, the MCA outputs an error message and closes the channel. Similarly, if during data transfer the exits in the send exit chain allocate more user space than they reserved such that fewer than 1024 bytes remain in the transmission buffer for message data, the MCA outputs an error message and closes

the channel. The limit of 1024 allows the control and administrative flows of the channel to be processed by the chain of send exits, without the need for the flows to be segmented.

This is an input/output field to the exit if *ExitReason* is MQXR_INIT, and an input field in all other cases. The field is not present if *Version* is less than MQCXP_VERSION_5.

SSLCertUserId (MQCHAR12)

This field specifies the UserId associated with the remote certificate.

It is blank on all platforms except z/OS

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6.

SSLRemCertIssNameLength (MQLONG)

This field specifies the length in bytes of the full Distinguished Name of the issuer of the remote certificate pointed to by SSLCertRemoteIssuerNamePtr.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6. The value is zero if it is not an SSL channel.

SSLRemCertIssNamePtr (PMQVOID)

This field specifies the address of the full Distinguished Name of the issuer of the remote certificate.

Its value is the null pointer if it is not an SSL channel.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6.

Note: The behavior of channel security exits in determining the Subject Distinguished Name and the Issuer Distinguished Name has altered in the release of IBM MQ v7.1. For more information see [Channel security exit programs](#).

SecurityParms (PMQCSP)

This field specifies the address of the MQCSP structure used to specify a user ID and password.

The initial value of this field is the null pointer.

This is an input/output field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6.

The value in this field that is returned by the exit must be usable by IBM MQ until MQXR_TERM.

CurHdrCompression (MQLONG)

This field specifies which technique is currently being used to compress the header data.

It is set to one of the following:

MQCOMPRESS_NONE

No header data compression is performed.

MQCOMPRESS_SYSTEM

Header data compression is performed.

The value can be altered by a sending channel's message exit to one of the negotiated supported values accessed from the HdrCompList field of the MQCD. This enables the technique used to compress the header data to be chosen for each message based on the content of the message. The altered value is used for the current message only. The channel ends if the attribute is altered to an unsupported value. The value is ignored if altered outside a sending channel's message exit.

This is an input/output field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6.

CurMsgCompression (MQLONG)

This field specifies which technique is currently being used to compress the message data.

It is set to one of the following:

MQCOMPRESS_NONE

No header data compression is performed.

MQCOMPRESS_RLE

Message data compression is performed using run-length encoding.

MQCOMPRESS_ZLIBFAST

Message data compression is performed using the zlib compression technique. A fast compression time is preferred.

MQCOMPRESS_ZLIBHIGH

Message data compression is performed using the zlib compression technique. A high level of compression is preferred.

The value can be altered by a sending channel's message exit to one of the negotiated supported values accessed from the `MsgCompList` field of the MQCD. This enables the technique used to compress the message data to be decided for each message based on the content of the message. The altered value is used for the current message only. The channel ends if the attribute is altered to an unsupported value. The value is ignored if altered outside a sending channel's message exit.

This is an input/output field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_6.

Hconn (MQHCONN)

This field specifies the connection handle that the exit uses if it needs to make any MQI calls within the exit.

This field is not relevant to exits running on client-connection channels, where it contains the value MQHC_UNUSABLE_HCONN (-1).

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_7.

SharingConversations (MQBOOL)

This field specifies whether the conversation is the only one that can currently be running on this channel instance, or whether more than one conversation can currently be running on this channel instance.

It also indicates whether the exit program is subject to the risk of the MQCD being altered by another exit program running at the same time.

This field is only relevant for exit programs running on client-connection or server-connection channels.

It is set to one of the following:

FALSE

The exit instance is the only exit instance that can currently be running on this channel instance. This allows the exit to safely update the MQCD fields without contention from other exits running on other channel instances. Whether changes to the MQCD fields are acted upon by the channel is defined by the table of MQCD fields in [“Changing MQCD fields in a channel exit” on page 1523](#).

TRUE

The exit instance is not the only exit instance that can currently be running on this channel instance. Any changes made to the MQCD are not acted upon by the channel, except for changes listed in the table of MQCD fields in [“Changing MQCD fields in a channel exit” on page 1523](#) for Exit Reasons other than MQXR_INIT. If this exit updates the MQCD fields, ensure there is no contention from other exits running on other conversations at the same time by providing serialization among the exits that run on this channel instance.

This is an input field to the exit. The field is not present if *Version* is less than MQCXP_VERSION_7.

MCAUserSource (MQLONG)

This field specifies the source of the provided MCA user ID.

It can contain one of the following values:

MQUSRC_MAP

The user ID is specified in the MCAUSER attribute.

MQUSRC_CHANNEL

The user ID is flowed from the inbound partner or specified in the MCAUSER field defined in the channel object.

This is an input field to the exit. The field is not present if Version is less than MQCXP_VERSION_8.

pEntryPoints (PMQIEP)

This field specifies the address of the interface entry point for the MQI or DCI call.

The field is not present if *Version* is less than MQCXP_VERSION_8.

RemoteProduct (MQCHAR4)

This field specifies the remote product name.

This field identifies the remote product of the client, for example, C or Java, as displayed in the **RPRODUCT** field of DISPLAY CHSATUS.

The field is not present if *Version* is less than MQCXP_VERSION_9.

RemoteVersion (MQCHAR8)

This field specifies the name of the remote version.

This field identifies the version of the client libraries, as displayed in the **RVERSION** field of DISPLAY CHSTATUS.

The field is not present if *Version* is less than MQCXP_VERSION_9.

C declaration

This declaration is the C declaration for the MQCXP structure.

```
typedef struct tagMQCXP MQCXP;
struct tagMQCXP {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     ExitId;           /* Type of exit */
    MQLONG     ExitReason;       /* Reason for invoking exit */
    MQLONG     ExitResponse;     /* Response from exit */
    MQLONG     ExitResponse2;    /* Secondary response from exit */
    MQLONG     Feedback;        /* Feedback code */
    MQLONG     MaxSegmentLength; /* Maximum segment length */
    MQBYTE16   ExitUserArea;     /* Exit user area */
    MQCHAR32   ExitData;        /* Exit data */
    MQLONG     MsgRetryCount;    /* Number of times the message has been
    MQLONG     MsgRetryInterval; /* Minimum interval in milliseconds after
                                which the put operation should be
                                retried */
    MQLONG     MsgRetryReason;   /* Reason code from previous attempt to
                                put the message */
    MQLONG     HeaderLength;     /* Length of header information */
    MQCHAR48   PartnerName;     /* Partner Name */
    MQLONG     FAPLevel;        /* Negotiated Formats and Protocols
                                level */
    MQLONG     CapabilityFlags;  /* Capability flags */
    MQLONG     ExitNumber;       /* Exit number */
    /* Ver:3 */
    /* Ver:4 */
    MQLONG     ExitSpace;        /* Number of bytes in transmission buffer
                                reserved for exit to use */
    /* Ver:5 */
    MQCHAR12   SSLCertUserid;    /* User identifier associated
                                with remote SSL certificate */
    MQLONG     SSLRemCertIssNameLength; /* Length of
                                distinguished name of issuer
                                of remote SSL certificate */
    MQPTR      SSLRemCertIssNamePtr; /* Address of
                                distinguished name of issuer
                                of remote SSL certificate */
    PMQVOID    SecurityParms;    /* Security parameters */
    MQLONG     CurHdrCompression; /* Header data compression
                                used for current message */
    MQLONG     CurMsgCompression; /* Message data compression
                                used for current message */
    /* Ver:6 */
```

```

MQHCONN    Hconn;                /* Connection handle */
MQBOOL     SharingConversations; /* Multiple conversations
                                possible on channel inst? */

/* Ver:7 */
MQLONG     MCAUserSource;        /* Source of the provided MCA user ID */
PMQIEP     pEntryPoints;        /* Address of the MQIEP structure */
/* Ver:8 */
MQCHAR4     RemoteProduct;       /* The identifier for the remote product */
MQCHAR8     RemoteVersion;       /* The version of the remote product */
/* Ver:9 */
};

```

COBOL declaration

This declaration is the COBOL declaration for the MQCXP structure.

```

**  MQCXP structure
10  MQCXP.
**    Structure identifier
15  MQCXP-STRUCID          PIC X(4).
**    Structure version number
15  MQCXP-VERSION         PIC S9(9) BINARY.
**    Type of exit
15  MQCXP-EXITID          PIC S9(9) BINARY.
**    Reason for invoking exit
15  MQCXP-EXITREASON      PIC S9(9) BINARY.
**    Response from exit
15  MQCXP-EXITRESPONSE    PIC S9(9) BINARY.
**    Secondary response from exit
15  MQCXP-EXITRESPONSE2   PIC S9(9) BINARY.
**    Feedback code
15  MQCXP-FEEDBACK        PIC S9(9) BINARY.
**    Maximum segment length
15  MQCXP-MAXSEGMENTLENGTH PIC S9(9) BINARY.
**    Exit user area
15  MQCXP-EXITUSERAREA    PIC X(16).
**    Exit data
15  MQCXP-EXITDATA        PIC X(32).
**    Number of times the message has been retried
15  MQCXP-MSGRETRYCOUNT  PIC S9(9) BINARY.
**    Minimum interval in milliseconds after which the put operation
**    should be retried
15  MQCXP-MSGRETRYINTERVAL PIC S9(9) BINARY.
**    Reason code from previous attempt to put the message
15  MQCXP-MSGRETRYREASON  PIC S9(9) BINARY.
**    Length of header information
15  MQCXP-HEADERLENGTH    PIC S9(9) BINARY.
**    Partner Name
15  MQCXP-PARTNERNAME     PIC X(48).
**    Negotiated Formats and Protocols level
15  MQCXP-FAPLEVEL        PIC S9(9) BINARY.
**    Capability flags
15  MQCXP-CAPABILITYFLAGS PIC S9(9) BINARY.
**    Exit number
15  MQCXP-EXITNUMBER      PIC S9(9) BINARY.
**    Number of bytes in transmission buffer reserved for exit to use
15  MQCXP-EXITSPEACE      PIC S9(9) BINARY.
**    User Id associated with remote certificate
15  MQCXP-SSLCERTUSERID   PIC X(12).
**    Length of distinguished name of issuer of remote SSL
**    certificate
15  MQCXP-SSLREMCERTISSNAMELENGTH PIC S9(9) BINARY.
**    Address of distinguished name of issuer of remote SSL
**    certificate
15  MQCXP-SSLREMCERTISSNAMEPTR  POINTER.
**    Security parameters
15  MQCXP-SECURITYPARMS    PIC S9(18) BINARY.
**    Header data compression used for current message
15  MQCXP-CURHDRCOMPRESSION PIC S9(9) BINARY.
**    Message data compression used for current message
15  MQCXP-CURMSGCOMPRESSION PIC S9(9) BINARY.
**    Connection handle
15  MQCXP-HCONN           PIC S9(9) BINARY.
**    Multiple conversations possible on channel instance?
15  MQCXP-SHARINGCONVERSATIONS PIC S9(9) BINARY.
**    Source of the provided MCA user ID
15  MQCXP-MCAUSERSOURCE   PIC S9(9) BINARY.

```

RPG declaration (ILE)

This declaration is the RPG declaration for the MQCXP structure.

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQCXP Structure
D*
D* Structure identifier
D CXSID 1 4
D* Structure version number
D CXVER 5 8I 0
D* Type of exit
D CXXID 9 12I 0
D* Reason for invoking exit
D CXREA 13 16I 0
D* Response from exit
D CXRES 17 20I 0
D* Secondary response from exit
D CXRE2 21 24I 0
D* Feedback code
D CXFB 25 28I 0
D* Maximum segment length
D CXMSL 29 32I 0
D* Exit user area
D CXUA 33 48
D* Exit data
D CXDAT 49 80
D* Number of times the message has been retrieved
D CXMRC 81 84I 0
D* Minimum interval in milliseconds after which the put operation
D* should be retried
D CXMRI 85 88I 0
D* Reason code from previous attempt to put the message
D CXMRR 89 92I 0
D* Length of header information
D CXHDL 93 96I 0
D* Partner Name
D CXPNM 97 144
D* Negotiated Formats and Protocols level
D CXFAP 145 148I 0
D* Capability flags
D CXCAP 149 152I 0
D* Exit number
D CXEXN 153 156I 0
D* Number of bytes in transmission buffer reserved for exit to use
D CXHDL 157 160I 0
D* User identifier associated with remote SSL certificate
D CXSSLCU 161 172
D* Length of distinguished name of issuer of remote SSL certificate
D CXSRCINL 173 176I 0
D* Address of distinguished name of issuer of remote SSL certificate
D CXSRCINP 177 192*
D* Security parameters
D CXSECP 193 208*
D* Header data compression used for current message
D CXCHC 209 212I 0
D* Message data compression used for current message
D CXCMC 213 216I 0
D* Connection handle
D CXHCONN 217 220I 0
D* Multiple conversations possible on channel instance?
D CXSHARECONV 221 224I 0
D* Source of the provided MCA user ID
D MCAUSERSOURCE 225 228I 0
```

System/390 assembler declaration

This declaration is the System/390 assembler declaration for the MQCXP structure.

MQCXP	DSECT	
MQCXP_STRUCID	DS	CL4 Structure identifier
MQCXP_VERSION	DS	F Structure version number
MQCXP_EXITID	DS	F Type of exit
MQCXP_EXITREASON	DS	F Reason for invoking exit
MQCXP_EXITRESPONSE	DS	F Response from exit
MQCXP_EXITRESPONSE2	DS	F Secondary response from exit
MQCXP_FEEDBACK	DS	F Feedback code
MQCXP_MAXSEGMENTLENGTH	DS	F Maximum segment length
MQCXP_EXITUSERAREA	DS	XL16 Exit user area
MQCXP_EXITDATA	DS	CL32 Exit data

MQCXP_MSGRETRYCOUNT	DS	F	Number of times the message has been retried
* MQCXP_MSGRETRYINTERVAL	DS	F	Minimum interval in milliseconds after which the put operation should be retried
* MQCXP_MSGRETRYREASON	DS	F	Reason code from previous attempt to put the message
* MQCXP_HEADERLENGTH	DS	F	Length of header information
MQCXP_PARTNERNAME	DS	CL48	Partner Name
MQCXP_FAPLEVEL	DS	F	Negotiated Formats and Protocols level
* MQCXP_CAPABILITYFLAGS	DS	F	Capability flags
MQCXP_EXITNUMBER	DS	F	Exit number
MQCXP_EXITSPLACE	DS	F	Number of bytes in transmission buffer reserved for exit to use
* MQCXP_SSLCERTUSERID	DS	CL12	User identifier associated with remote SSL certificate
* MQCXP_SSLREMCERTISSNAMELENGTH	DS	F	Length of distinguished name of issuer of remote SSL certificate
* MQCXP_SSLREMCERTISSNAMEPTR	DS	F	Address of distinguished name of issuer of remote SSL certificate
* MQCXP_SECURITYPARMS	DS	F	Address of security parameters
MQCXP_CURHDRCOMPRESSION	DS	F	Header data compression used for current message
* MQCXP_CURMSGCOMPRESSION	DS	F	Message data compression used for current message
* MQCXP_HCONN	DS	F	Connection handle
MQCXP_SHARINGCONVERSATIONS	DS	F	Multiple conversations possible on channel inst?
* MQCXP_MCAUSERSOURCE	DS	F	Source of the provided MCA user ID
MQCXP_LENGTH	EQU	*-MQCXP	
	ORG	MQCXP	
MQCXP_AREA	DS	CL (MQCXP_LENGTH)	

MQXWD - Exit wait descriptor

The MQXWD structure is an input/output parameter on the MQXWAIT call.

This structure is supported only on z/OS.

Related reference

[“Fields” on page 1540](#)

This topic lists all the fields in the MQXWD structure and describes each field.

[“C declaration” on page 1541](#)

This declaration is the C declaration for the MQXWD structure.

[“System/390 assembler declaration” on page 1541](#)

This declaration is the System/390 assembler declaration for the MQXWD structure.

Fields

This topic lists all the fields in the MQXWD structure and describes each field.

StrucId (MQCHAR4)

This field specifies the structure identifier.

The value must be:

MQXWD_STRUC_ID

Identifier for exit wait descriptor structure.

For the C programming language, the constant MQXWD_STRUC_ID_ARRAY is also defined; this constant has the same value as MQXWD_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQXWD_STRUC_ID.

Version (MQLONG)

This field specifies the structure version number.

The value must be:

MQXWD_VERSION_1

Version number for exit wait descriptor structure.

The initial value of this field is MQXWD_VERSION_1.

Reserved1 (MQLONG)

This field is reserved. Its value must be zero.

This is an input field.

Reserved2 (MQLONG)

This field is reserved. Its value must be zero.

This is an input field.

Reserved3 (MQLONG)

This field is reserved. Its value must be zero.

This is an input field.

ECB (MQLONG)

This field specifies the event control block to wait on.

This field is the event control block (ECB) to wait on. It must be set to zero before the MQXWAIT call is issued; on successful completion it contains the post code.

This field is an input/output field.

C declaration

This declaration is the C declaration for the MQXWD structure.

```
typedef struct tagMQXWD MQXWD;
struct tagMQXWD {
    MQCHAR4  StrucId;    /* Structure identifier */
    MQLONG   Version;    /* Structure version number */
    MQLONG   Reserved1;  /* Reserved */
    MQLONG   Reserved2;  /* Reserved */
    MQLONG   Reserved3;  /* Reserved */
    MQLONG   ECB;        /* Event control block to wait on */
};
```

System/390 assembler declaration

This declaration is the System/390 assembler declaration for the MQXWD structure.

MQXWD	DSECT		
MQXWD_STRUCID	DS	CL4	Structure identifier
MQXWD_VERSION	DS	F	Structure version number
MQXWD_RESERVED1	DS	F	Reserved
MQXWD_RESERVED2	DS	F	Reserved
MQXWD_RESERVED3	DS	F	Reserved
MQXWD_ECB	DS	F	Event control block to wait on
*			
MQXWD_LENGTH	EQU	*-MQXWD	
	ORG	MQXWD	
MQXWD_AREA	DS	CL(MQXWD_LENGTH)	

Cluster workload exit call and data structures

This section provides reference information for the cluster workload exit and the data structures. This is general-use programming interface information.


You can write cluster workload exits in the following programming languages:

- C
- System/390 assembler (IBM MQ for z/OS)

The call is described in:

- [“MQ_CLUSTER_WORKLOAD_EXIT - Call description” on page 1542](#)

The structure data types used by the exit are described in:

- [“MQXCLWLN - Navigate Cluster workload records” on page 1544](#)
- [“MQWXP - Cluster workload exit parameter structure” on page 1547](#)
- [“MQWDR - Cluster workload destination record structure” on page 1555](#)
- [“MQWQR - Cluster workload queue record structure” on page 1559](#)
- [“MQWCR - Cluster workload cluster record structure” on page 1564](#)
-  [Asynchronous behavior of CLUSTER commands on z/OS](#)

Throughout this section, queue-manager attributes and queue attributes are shown in full. The equivalent names that are used in the MQSC commands are shown below. For details of MQSC commands, see [MQSC commands](#).

Table 644. Queue-manager attributes	
Full name	Name used in MQSC
<i>ClusterWorkloadData</i>	CLWLDATA
<i>ClusterWorkloadExit</i>	CLWLEXIT
<i>ClusterWorkloadLength</i>	CLWLEN

Table 645. Queue attributes	
Full name	Name used in MQSC
<i>DefBind</i>	DEFBIND
<i>DefPersistence</i>	DEFPSIST
<i>DefPriority</i>	DEFPRTY
<i>InhibitPut</i>	PUT
<i>QDesc</i>	DESCR

Related tasks

[Writing and compiling cluster workload exits](#)

MQ_CLUSTER_WORKLOAD_EXIT - Call description

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

Note: No entry point called MQ_CLUSTER_WORKLOAD_EXIT is provided by the queue manager. Instead, the name of the cluster workload exit is defined by the ClusterWorkloadExit queue-manager attribute.

The MQ_CLUSTER_WORKLOAD_EXIT exit is supported on all platforms.

Syntax

```
MQ_CLUSTER_WORKLOAD_EXIT (ExitParms)
```

Related reference

[MQXCLWLN - Navigate Cluster workload records](#)

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

[MQWXP - Cluster workload exit parameter structure](#)

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

[MQWDR - Cluster workload destination record structure](#)

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

[MQWQR - Cluster workload queue record structure](#)

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

[MQWCR - Cluster workload cluster record structure](#)

The following table summarizes the fields in the MQWCR cluster workload record structure.

Parameters for MQ_CLUSTER_WORKLOAD_EXIT

Description of the parameters in the MQ_CLUSTER_WORKLOAD_EXIT call.

ExitParms (MQWXP) - input/output

Exit parameter block.

- The exit sets information in MQWXP to indicate how to manage the workload.

Related reference

[Usage notes](#)

The function performed by the cluster workload exit is defined by the provider of the exit. The exit, however, must conform to the rules defined in the associated control block MQWXP.

[Language invocations for MQ_CLUSTER_WORKLOAD_EXIT](#)

The MQ_CLUSTER_WORKLOAD_EXIT supports two languages, C and High Level Assembler.

Usage notes

The function performed by the cluster workload exit is defined by the provider of the exit. The exit, however, must conform to the rules defined in the associated control block MQWXP.

No entry point called MQ_CLUSTER_WORKLOAD_EXIT is provided by the queue manager. However, a typedef is provided for the name MQ_CLUSTER_WORKLOAD_EXIT in the C programming language. Use the typedef to declare the user-written exit, to ensure that the parameters are correct.

Related reference

[Parameters for MQ_CLUSTER_WORKLOAD_EXIT](#)

Description of the parameters in the MQ_CLUSTER_WORKLOAD_EXIT call.

[Language invocations for MQ_CLUSTER_WORKLOAD_EXIT](#)

The MQ_CLUSTER_WORKLOAD_EXIT supports two languages, C and High Level Assembler.

Language invocations for MQ_CLUSTER_WORKLOAD_EXIT

The MQ_CLUSTER_WORKLOAD_EXIT supports two languages, C and High Level Assembler.

C invocation

```
MQ_CLUSTER_WORKLOAD_EXIT (&ExitParms);
```

Replace *MQ_CLUSTER_WORKLOAD_EXIT* with the name of your cluster workload exit function.

Declare the *MQ_CLUSTER_WORKLOAD_EXIT* parameters as follows:

```
MQWXP ExitParms; /* Exit parameter block */
```

High Level Assembler invocation

```
CALL EXITNAME, (EXITPARMS)
```

Declare the parameters as follows:

EXITPARMS	CMQWXP	Exit parameter block
-----------	--------	----------------------

Related reference

[Parameters for MQ_CLUSTER_WORKLOAD_EXIT](#)

Description of the parameters in the MQ_CLUSTER_WORKLOAD_EXIT call.

Usage notes

The function performed by the cluster workload exit is defined by the provider of the exit. The exit, however, must conform to the rules defined in the associated control block MQWXP.

MQXCLWLN - Navigate Cluster workload records

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

The cluster cache is an area of main storage used to store information relating to the cluster.

If the cluster cache is static, it has a fixed size. If you set it to dynamic, the cluster cache can expand as required.

Set the type of cluster cache to STATIC or DYNAMIC using either a system parameter or macro.

- The system parameter ClusterCacheType on platforms other than z/OS
- The CLCACHE parameter in the CSQ6SYSP macro on z/OS.

Syntax

```
MQXCLWLN (ExitParms, CurrentRecord, NextOffset, NextRecord, Compcode, Reason)
```

Related reference

[MQ_CLUSTER_WORKLOAD_EXIT - Call description](#)

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

[MQWXP - Cluster workload exit parameter structure](#)

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

[MQWDR - Cluster workload destination record structure](#)

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

[MQWQR - Cluster workload queue record structure](#)

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

[MQWCR - Cluster workload cluster record structure](#)

The following table summarizes the fields in the MQWCR cluster workload record structure.

Parameters for MQXCLWLN - Navigate Cluster workload records

Description of the parameters in the MQXCLWLN call.

ExitParms (MQWXP) - input/output

Exit parameter block.

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate how to manage the workload.

CurrentRecord (MQPTR) - input

Address of current record.

This structure contains information relating to the address of the record currently being examined by the exit. The record must be one of the following types:

- Cluster workload destination record (MQWDR)
- Cluster workload queue record (MQWQR)

- Cluster workload cluster record (MQWCR)

***NextOffset* (MQLONG) - input**

Offset of next record.

This structure contains information relating to the offset of the next record or structure. *NextOffset* is the value of the appropriate offset field in the current record, and must be one of the following fields:

- ChannelDefOffset field in MQWDR
- ClusterRecOffset field in MQWDR
- ClusterRecOffset field in MQWQR
- ClusterRecOffset field in MQWCR

***NextRecord* (MQPTR) - output**

Address of next record or structure.

This structure contains information relating to the address of the next record or structure.

If *CurrentRecord* is the address of an MQWDR, and *NextOffset* is the value of the ChannelDefOffset field, *NextRecord* is the address of the channel definition structure (MQCD).

If there is no next record or structure, the queue manager sets *NextRecord* to the null pointer, and the call returns completion code MQCC_WARNING and reason code MQRC_NO_RECORD_AVAILABLE.

***CompCode* (MQLONG) - output**

Completion code.

The completion code has one of the following values:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

***Reason* (MQLONG) - output**

Reason code qualifying CompCode

If CompCode is MQCC_OK:

MQRC_NONE

(0, X'0000')

No reason to report.

If CompCode is MQCC_WARNING:

MQRC_NO_RECORD_AVAILABLE

(2359, X'0937')

No record available. An MQXCLWLN call was issued from a cluster workload exit to obtain the address of the next record in the chain. The current record is the last record in the chain.

Corrective action: None.

If CompCode is MQCC_FAILED:

MQRC_CURRENT_RECORD_ERROR

(2357, X'0935')

CurrentRecord parameter not valid. An MQXCLWLN call was issued from a cluster workload exit to obtain the address of the next record in the chain. The address specified by the *CurrentRecord* parameter is not the address of a valid record.

CurrentRecord must be the address of a destination record, MQWDR, queue record (MQWQR), or cluster record (MQWCR) residing within the cluster cache. Corrective action: Ensure that the cluster workload exit passes the address of a valid record residing in the cluster cache.

MQRC_ENVIRONMENT_ERROR
(2012, X'07DC')

Call not valid in environment. An MQXCLWLN call was issued, but not from a cluster workload exit.

MQRC_NEXT_OFFSET_ERROR
(2358, X'0936')

NextOffset parameter not valid. An MQXCLWLN call was issued from a cluster workload exit to obtain the address of the next record in the chain. The offset specified by the *NextOffset* parameter is not valid. *NextOffset* must be the value of one of the following fields:

- ChannelDefOffset field in MQWDR
- ClusterRecOffset field in MQWDR
- ClusterRecOffset field in MQWQR
- ClusterRecOffset field in MQWCR

Corrective action: Ensure that the value specified for the *NextOffset* parameter is the value of one of the fields listed previously.

MQRC_NEXT_RECORD_ERROR
(2361, X'0939')

NextRecord parameter not valid.

MQRC_WXP_ERROR
(2356, X'0934')

Workload exit parameter structure not valid. An MQXCLWLN call was issued from a cluster workload exit to obtain the address of the next record in the chain. The workload exit parameter structure *ExitParms* is not valid, for one of the following reasons:

- The parameter pointer is not valid. It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.
- The StrucId field is not MQWXP_STRUC_ID.
- The Version field is not MQWXP_VERSION_2.
- The Context field does not contain the value passed to the exit by the queue manager.

Corrective action: Ensure that the parameter specified for *ExitParms* is the MQWXP structure that was passed to the exit when the exit was invoked.

Related reference

[Usage notes for MQXCLWLN - Navigate Cluster workload records](#)

Use MQXCLWLN to navigate through cluster records, even if the cache is static.

[Language invocations of MQXCLWLN](#)

MQXCLWLN supports two languages, C and High Level Assembler.

Usage notes for MQXCLWLN - Navigate Cluster workload records

Use MQXCLWLN to navigate through cluster records, even if the cache is static.

If the cluster cache is dynamic, the MQXCLWLN call must be used to navigate through the records. The exit ends abnormally if simple pointer-and-offset arithmetic is used to navigate through the records.

If the cluster cache is static, MQXCLWLN need not be used to navigate through the records. Typically use MQXCLWLN even when the cache is static. You can then change the cluster cache to being dynamic without needing to change the workload exit.

Related reference

[Parameters for MQXCLWLN - Navigate Cluster workload records](#)

Description of the parameters in the MQXCLWLN call.

[Language invocations of MQXCLWLN](#)

MQXCLWLN supports two languages, C and High Level Assembler.

Language invocations of MQXCLWLN

MQXCLWLN supports two languages, C and High Level Assembler.

C invocation

```
MQXCLWLN (&ExitParms, CurrentRecord, NextOffset, &NextRecord, &CompCode, &Reason) ;
```

Declare the parameters as follows:

```
typedef struct tagMQXCLWLN {
MQWXP   ExitParms;      /* Exit parameter block */
MQPTR   CurrentRecord;  /* Address of current record*/
MQLONG  NextOffset;     /* Offset of next record */
MQPTR   NextRecord;     /* Address of next record or structure */
MQLONG  CompCode;       /* Completion code */
MQLONG  Reason;         /* Reason code qualifying CompCode */
}
```

High Level Assembler invocation

```
CALL MQXCLWLN, (CLWLEXITPARMS, CURRENTRECORD, NEXTOFFSET, NEXTRECORD, COMPCODE, REASON)
```

Declare the parameters as follows:

```
CLWLEXITPARMS CMQWXP, Cluster workload exit parameter block
CURRENTRECORD CMQWDRA, Current record
NEXTOFFSET     DS F      Next offset
NEXTRECORD     DS F      Next record
COMPCODE       DS F      Completion code
REASON         DS F      Reason code qualifying COMPCODE
```

Related reference

[Parameters for MQXCLWLN - Navigate Cluster workload records](#)

Description of the parameters in the MQXCLWLN call.

[Usage notes for MQXCLWLN - Navigate Cluster workload records](#)

Use MQXCLWLN to navigate through cluster records, even if the cache is static.

MQWXP - Cluster workload exit parameter structure

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

Table 646. Fields in MQWXP		
Field	Description	Page
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>ExitId</i>	Type of exit	ExitId
<i>ExitReason</i>	Reason for invoking exit	ExitReason
<i>ExitResponse</i>	Response from exit	ExitResponse
<i>ExitResponse2</i>	Secondary response from exit	ExitResponse2
<i>Feedback</i>	Feedback code	Feedback
<i>Flags</i>	Flags values. These bit flags are used to indicate information about the message being put	Flags
<i>ExitUserArea</i>	Exit user area	ExitUserArea

Table 646. Fields in MQWXP (continued)		
Field	Description	Page
<i>ExitData</i>	Exit data	ExitData
<i>MsgDescPtr</i>	Address of message descriptor (MQMD)	MsgDescPtr
<i>MsgBufferPtr</i>	Address of buffer containing some or all the message data	MsgBufferPtr
<i>MsgBufferLength</i>	Length of buffer containing message data	MsgBufferLength
<i>MsgLength</i>	Length of complete message	MsgLength
<i>QName</i>	Name of queue	QName
<i>QMgrName</i>	Name of local queue manager	QMgrName
<i>DestinationCount</i>	Number of possible destinations	DestinationCount
<i>DestinationChosen</i>	Destination chosen	DestinationChosen
<i>DestinationArrayPtr</i>	Address of an array of pointers to destination records (MQWDR)	DestinationArrayPtr
<i>QArrayPtr</i>	Address of an array of pointers to queue records (MQWQR)	QArrayPtr
Note: The remaining fields are ignored if Version is less than MQWXP_VERSION_2.		
<i>CacheContext</i>	Context information	CacheContext
<i>CacheType</i>	Type of cluster cache	CacheType
Note: The remaining fields are ignored if Version is less than MQWXP_VERSION_3.		
<i>CLWLMRUChannels</i>	Maximum number of allowed active outbound cluster channels	CLWLMRUChannels
Note: The remaining fields are ignored if Version is less than MQWXP_VERSION_4.		
<i>pEntryPoints</i>	Address of the MQIEP structure to allow MQI and DCI calls to be made	pEntryPoints

The cluster workload exit parameter structure describes the information that is passed to the cluster workload exit.

The cluster workload exit parameter structure is supported on all platforms

Additionally, the MQWXP1, MQWXP2 and MQWXP3 structures are available for backwards compatibility.

Related reference

[MQ_CLUSTER_WORKLOAD_EXIT - Call description](#)

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

[MQXCLWLN - Navigate Cluster workload records](#)

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

[MQWDR - Cluster workload destination record structure](#)

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

[MQWQR - Cluster workload queue record structure](#)

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

MQWCR - Cluster workload cluster record structure

The following table summarizes the fields in the MQWCR cluster workload record structure.

Fields in MQWXP - Cluster workload exit parameter structure

Description of the fields in the MQWXP - Cluster workload exit parameter structure

StrucId (MQCHAR4) - input

The structure identifier for the cluster workload exit parameter structure.

- The StrucId value is MQWXP_STRUC_ID.
- For the C programming language, the constant MQWXP_STRUC_ID_ARRAY is also defined. It has the same value as MQWXP_STRUC_ID. It is an array of characters instead of a string.

Version (MQLONG) - input

Indicates the structure version number. Version takes one of the following values:

MQWXP_VERSION_1

Version-1 cluster workload exit parameter structure.

MQWXP_VERSION_1 is supported in all environments.

MQWXP_VERSION_2

Version-2 cluster workload exit parameter structure.

MQWXP_VERSION_2 is supported in the following environments: AIX, HP-UX, Linux, IBM i, Solaris and Windows.

MQWXP_VERSION_3

Version-3 cluster workload exit parameter structure.

MQWXP_VERSION_3 is supported in the following environments: AIX, HP-UX, Linux, IBM i, Solaris and Windows.

MQWXP_VERSION_4

Version-4 cluster workload exit parameter structure.

MQWXP_VERSION_4 is supported in the following environments: AIX, HP-UX, Linux, IBM i, Solaris and Windows.

MQWXP_CURRENT_VERSION

Current version of cluster workload exit parameter structure.

ExitId (MQLONG) - input

Indicates the type of exit being called. The cluster workload exit is the only supported exit.

- The ExitId value must be MQXT_CLUSTER_WORKLOAD_EXIT

ExitReason (MQLONG) - input

Indicates the reason for invoking the cluster workload exit. ExitReason takes one of the following values:

MQXR_INIT

Indicates that the exit is being invoked for the first time.

Acquire and initialize any resources that the exit might need, such as main storage.

MQXR_TERM

Indicates that the exit is about to be terminated.

Free any resources that the exit might have acquired since it was initialized, such as main storage.

MQXR_CLWL_OPEN

Called by MQOPEN.

MQXR_CLWL_PUT

Called by MQPUT or MQPUT1.

MQXR_CLWL_MOVE

Called by MCA when the channel state has changed.

MQXR_CLWL_REPOS

Called by MQPUT or MQPUT1 for a repository-manager PCF message.

MQXR_CLWL_REPOS_MOVE

Called by MCA for a repository-manager PCF message if the channel state has changed.

ExitResponse (MQLONG) - output

Set ExitResponse to indicate whether processing of the message continues. It must be one of the following values:

MQXCC_OK

Continue processing the message normally.

- DestinationChosen identifies the destination to which the message is to be sent.

MQXCC_SUPPRESS_FUNCTION

Discontinue processing the message.

- The actions taken by the queue manager depend on the reason the exit was invoked:

Table 647. Actions taken by the queue manager.

There are two columns in this table. The first column lists the exit reasons and the second column describes the action taken for each exit reason.

ExitReason	Action taken
<ul style="list-style-type: none"> – MQXR_CLWL_OPEN – MQXR_CLWL_REPOS – MQXR_CLWL_PUT 	MQOPEN, MQPUT, or MQPUT1 call fail with completion code MQCC_FAILED and reason code MQRC_STOPPED_BY_CLUSTER_EXIT.
<ul style="list-style-type: none"> – MQXR_CLWL_MOVE – MQXR_CLWL_REPOS_MOVE 	The message is placed on the dead-letter queue.

MQXCC_SUPPRESS_EXIT

Continue processing the current message normally. Do not invoke the exit again until the queue manager shuts down.

The queue manager processes subsequent messages as if the ClusterWorkloadExit queue-manager attribute is blank. DestinationChosen identifies the destination to which the current message is sent.

Any other value

Process the message as if MQXCC_SUPPRESS_FUNCTION is specified.

ExitResponse2 (MQLONG) - input/output

Set ExitResponse2 to provide the queue manager with more information.

- MQXR2_STATIC_CACHE is the default value, and is set on entry to the exit.
- When ExitReason has the value MQXR_INIT, the exit can set one of the following values in ExitResponse2:

MQXR2_STATIC_CACHE

The exit requires a static cluster cache.

- If the cluster cache is static, the exit need not use the MQXCLWLN call to navigate the chains of records in the cluster cache.
- If the cluster cache is dynamic, the exit cannot navigate correctly through the records in the cache.

Note: The queue manager processes the return from the MQXR_INIT call as though the exit had returned MQXCC_SUPPRESS_EXIT in the ExitResponse field.

MQXR2_DYNAMIC_CACHE

The exit can operate with either a static or dynamic cache.

- If the exit returns this value, the exit must use the MQXCLWLN call to navigate the chains of records in the cluster cache.

Feedback (MQLONG) - input

A reserved field. The value is zero.

Flags (MQLONG) - input

Indicates information about the message being put.

- The value of Flags is MQWXP_PUT_BY_CLUSTER_CHL. The message originates from a cluster channel, rather than locally or from a non-cluster channel. In other words, the message has come from another cluster queue manager.

Reserved (MQLONG) - input

A reserved field. The value is zero.

ExitUserArea (MQBYTE16) - input/output

Set ExitUserArea to communicate between calls to the exit.

- ExitUserArea is initialized to binary zero before the first invocation of the exit. Any changes made to this field by the exit are preserved across the invocations of the exit that occur between the MQCONN call and the matching MQDISC call. The field is reset to binary zero when the MQDISC call occurs.
- The first invocation of the exit is indicated by the ExitReason field having the value MQXR_INIT.
- The following constants are defined:

MQXUA_NONE - string

MQXUA_NONE_ARRAY - character array

No user information. Both constants are binary zero for the length of the field.

MQ_EXIT_USER_AREA_LENGTH

The length of ExitUserArea.

ExitData (MQCHAR32) - input

The value of the ClusterWorkloadData queue-manager attribute. If no value has been defined for that attribute, this field is all blanks.

- The length of ExitData is given by MQ_EXIT_DATA_LENGTH.

MsgDescPtr (PMQMD) - input

The address of a copy of the message descriptor (MQMD) for the message being processed.

- Any changes made to the message descriptor by the exit are ignored by the queue manager.
- If ExitReason has one of the following values MsgDescPtr is set to the null pointer, and no message descriptor is passed to the exit:
 - MQXR_INIT
 - MQXR_TERM
 - MQXR_CLWL_OPEN

MsgBufferPtr (PMQVOID) - input

The address of a buffer containing a copy of the first MsgBufferLength bytes of the message data.

- Any changes made to the message data by the exit are ignored by the queue manager.
- No message data is passed to the exit when:
 - MsgDescPtr is the null pointer.
 - The message has no data.
 - The ClusterWorkloadLength queue-manager attribute is zero.

In these cases, MsgBufferPtr is the null pointer.

MsgBufferLength (MQLONG) - input

The length of the buffer containing the message data passed to the exit.

- The length is controlled by the ClusterWorkloadLength queue-manager attribute.
- The length might be less than the length of the complete message, see MsgLength.

MsgLength (MQLONG) - input

The length of the complete message passed to the exit.

- MsgBufferLength might be less than the length of the complete message.
- MsgLength is zero if ExitReason is MQXR_INIT, MQXR_TERM, or MQXR_CLWL_OPEN.

QName (MQCHAR48) - input

The name of the destination queue. The queue is a cluster queue.

- The length of QName is MQ_Q_NAME_LENGTH.

QMgrName (MQCHAR48) - input

The name of the local queue manager that has invoked the cluster workload exit.

- The length of QMgrName is MQ_Q_MGR_NAME_LENGTH.

DestinationCount (MQLONG) - input

The number of possible destinations. Destinations are instances of the destination queue and are described by destination records.

- A destination record is a MQWDR structure. There is one structure for each possible route to each instance of the queue.
- MQWDR structures are addressed by an array of pointers, see DestinationArrayPtr.

DestinationChosen (MQLONG) - input/output

The chosen destination.

- The number of the MQWDR structure that identifies the route and queue instance where the message is to be sent.
- The value is in the range 1 - DestinationCount.
- On input to the exit, DestinationChosen indicates the route and queue instance that the queue manager has selected. The exit can accept this choice, or choose a different route and queue instance.
- The value set by the exit must be in the range 1 - DestinationCount. If any other value is returned, the queue manager uses the value of DestinationChosen on input to the exit.

DestinationArrayPtr (PPMQWDR) - input

The address of an array of pointers to destination records (MQWDR).

- There are DestinationCount destination records.

QArrayPtr (PPMQWQR) - input

The address of an array of pointers to queue records (MQWQR).

- If queue records are available, there are DestinationCount of them.
- If no queue records are available, QArrayPtr is the null pointer.

Note: QArrayPtr can be the null pointer even when DestinationCount is greater than zero.

CacheContext (MQPTR) : Version 2 - input

The CacheContext field is reserved for use by the queue manager. The exit must not alter the value of this field.

CacheType (MQLONG) : Version 2 - input

The cluster cache has one of the following types:

MQCLCT_STATIC

The cache is static.

- The size of the cache is fixed, and cannot grow as the queue manager operates.
- You do not need to use the MQXCLWLN call to navigate the records in this type of cache.

MQCLCT_DYNAMIC

The cache is dynamic.

- The size of the cache can increase in order to accommodate the varying cluster information.
- You must use the MQXCLWLN call to navigate the records in this type of cache.

CLWLMRUChannels (MQLONG) : Version 3 - input

Indicates the maximum number of active outbound cluster channels, to be considered for use by the cluster workload choice algorithm.

- CLWLMRUChannels is a value 1 - 999 999 999.

pEntryPoints (PMQIEP) : Version 4

The address of an MQIEP structure through which MQI and DCI calls can be made.

Related reference

[Initial values and language declarations for MQWXP](#)

Initial values and C and High Level Assembler Language declarations for MQWXP - Cluster workload exit parameter structure.

Initial values and language declarations for MQWXP

Initial values and C and High Level Assembler Language declarations for MQWXP - Cluster workload exit parameter structure.

Table 648. Initial values of fields in MQWXP		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQWXP_STRUC_ID	'WXP'
<i>Version</i>	MQWXP_VERSION_2	2
<i>ExitId</i>	None	0
<i>ExitReason</i>	MQXCC_OK	0
<i>ExitResponse</i>	None	0
<i>ExitResponse2</i>	None	0
<i>Flags</i>	None	0
<i>ExitUserArea</i>	{MQXUA_NONE_ARRAY}	0
<i>ExitData</i>	None	" "
<i>MsgDescPtr</i>	None	NULL
<i>MsgBufferPtr</i>	None	NULL
<i>MsgBufferLength</i>	None	0
<i>MsgBufferPtr</i>	None	0
<i>QName</i>	None	" "
<i>QMgrName</i>	None	" "
<i>DestinationCount</i>	None	0
<i>DestinationChosen</i>	None	0
<i>DestinationArrayPtr</i>	None	NULL
<i>QArrayPtr</i>	None	NULL
<i>CacheContext</i>	None	NULL

Table 648. Initial values of fields in MQWXP (continued)

Field name	Name of constant	Value of constant
<i>CacheType</i>	MQCLCT_DYNAMIC	1
<i>CLWLMRUChannels</i>	None	0
<i>pEntryPoints</i>	None	NULL

Notes:

1. The symbol ~ represents a single blank character.
2. In the C programming language, the macro variable MQWXP_DEFAULT contains the default values. Use it in the following way to provide initial values for the fields in the structure:

```
MQWDR MyWXP = {MQWXP_DEFAULT};
```

C declaration

```
typedef struct tagMQWXP {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;           /* Structure version number */
    MQLONG    ExitId;            /* Type of exit */
    MQLONG    ExitReason;        /* Reason for invoking exit */
    MQLONG    ExitResponse;      /* Response from exit */
    MQLONG    ExitResponse2;     /* Reserved */
    MQLONG    Feedback;         /* Reserved */
    MQLONG    Flags;             /* Flags */
    MQBYTE16  ExitUserArea;      /* Exit user area */
    MQCHAR32  ExitData;          /* Exit data */
    PMQMD     MsgDescPtr;        /* Address of message descriptor */
    PMQVOID   MsgBufferPtr;      /* Address of buffer containing some
                                or all of the message data */
    MQLONG    MsgBufferLength;   /* Length of buffer containing message
                                data */
    MQLONG    MsgLength;         /* Length of complete message */
    MQCHAR48  QName;             /* Queue name */
    MQCHAR48  QMgrName;          /* Name of local queue manager */
    MQLONG    DestinationCount;  /* Number of possible destinations */
    MQLONG    DestinationChosen; /* Destination chosen */
    PPMQWDR   DestinationArrayPtr; /* Address of an array of pointers to
                                destination records */
    PPMQWQR   QArrayPtr;         /* Address of an array of pointers to
                                queue records */

    /* version 1 */
    MQPTR     CacheContext;      /* Context information */
    MQLONG    CacheType;         /* Type of cluster cache */
    /* version 2 */
    MQLONG    CLWLMRUChannels;   /* Maximum number of most recently
                                used cluster channels */

    /* version 3 */
    PMQIEP    pEntryPoints;      /* Address of the MQIEP structure */
    /* version 4 */
};
```

High Level Assembler

MQWXP	DSECT	
MQWXP_STRUCID	DS	CL4 Structure identifier
MQWXP_VERSION	DS	F Structure version number
MQWXP_EXITID	DS	F Type of exit
MQWXP_EXITREASON	DS	F Reason for invoking exit
MQWXP_EXITRESPONSE	DS	F Response from exit
MQWXP_EXITRESPONSE2	DS	F Reserved
MQWXP_FEEDBACK	DS	F Reserved
MQWXP_RESERVED	DS	F Reserved
MQWXP_EXITUSERAREA	DS	XL16 Exit user area
MQWXP_EXITDATA	DS	CL32 Exit data
MQWXP_MSGDESCPTR	DS	F Address of message
*		descriptor
MQWXP_MSGBUFFERPTR	DS	F Address of buffer containing

*				some or all of the message data
*				data
MQWXP_MSGBUFFERLENGTH	DS	F		Length of buffer containing message data
*				Length of complete message
MQWXP_MSGLENGTH	DS	F		Queue name
MQWXP_QNAME	DS	CL48		Name of local queue manager
MQWXP_QMGRNAME	DS	CL48		Number of possible destinations
MQWXP_DESTINATIONCOUNT	DS	F		Destination chosen
*				Address of an array of pointers to destination records
MQWXP_DESTINATIONCHOSEN	DS	F		Address of an array of pointers to queue records
MQWXP_DESTINATIONARRAYPTR	DS	F		Context information
*				Type of cluster cache
MQWXP_QARRAYPTR	DS	F		Number of most recently used channels for workload balancing
*				
MQWXP_CACHECONTEXT	DS	F		
MQWXP_CACHETYPE	DS	F		
MQWXP_CLWLMRCHANNELS	DS	F		
*				
MQWXP_LENGTH	EQU	*-MQWXP		Length of structure
	ORG	MQWXP		
MQWXP_AREA	DS	CL(MQWXP_LENGTH)		

Related reference

[Fields in MQWXP - Cluster workload exit parameter structure](#)

[Description of the fields in the MQWXP - Cluster workload exit parameter structure](#)

MQWDR - Cluster workload destination record structure

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

Table 649. Fields in MQWDR		
Field	Description	Page
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQWDR structure	StrucLength
<i>QMgrFlags</i>	Queue-manager flags	QMgrFlags
<i>QMgrIdentifier</i>	Queue-manager identifier	QMgrIdentifier
<i>QMgrName</i>	Queue-manager name	QMgrName
<i>ClusterRecOffset</i>	Logical offset of first cluster record (MQWCR)	ClusterRecOffset
<i>ChannelState</i>	Channel state	ChannelState
<i>ChannelDefOffset</i>	Logical offset of channel-definition structure (MQCD)	ChannelDefOffset
Note: The remaining fields are ignored if Version is less than MQWDR_VERSION_2.		
<i>DestSeqNumber</i>	Channel destination sequence number	DestSeqNumber
<i>DestSeqFactor</i>	Channel destination sequence factor for weighting	DestSeqFactor

The cluster workload destination record structure contains information relating to one of the possible destinations for the message. There is one cluster workload destination record structure for each instance of the destination queue.

The cluster workload destination record structure is supported in all environments.

Additionally, the MQWDR1 and MQWDR2 structures are available for backwards compatibility.

Related reference

[MQ_CLUSTER_WORKLOAD_EXIT - Call description](#)

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

MQXCLWLN - Navigate Cluster workload records

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

MQWXP - Cluster workload exit parameter structure

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

MQWQR - Cluster workload queue record structure

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

MQWCR - Cluster workload cluster record structure

The following table summarizes the fields in the MQWCR cluster workload record structure.

Fields in MQWDR - Cluster workload destination record structure

Description of the parameters in the MQWDR - Cluster workload destination record structure.

StrucId (MQCHAR4) - input

The structure identifier for the cluster workload destination record structure.

- The StrucId value is MQWDR_STRUC_ID.
- For the C programming language, the constant MQWDR_STRUC_ID_ARRAY is also defined. It has the same value as MQWDR_STRUC_ID. It is an array of characters instead of a string.

Version (MQLONG) - input

The structure version number. Version takes one of the following values:

MQWDR_VERSION_1

Version-1 cluster workload destination record.

MQWDR_VERSION_2

Version-2 cluster workload destination record.

MQWDR_CURRENT_VERSION

Current version of cluster workload destination record.

StrucLength (MQLONG) - input

The length of MQWDR structure. StrucLength takes one of the following values:

MQWDR_LENGTH_1

Length of version-1 cluster workload destination record.

MQWDR_LENGTH_2

Length of version-2 cluster workload destination record.

MQWDR_CURRENT_LENGTH

Length of current version of cluster workload destination record.

QMgrFlags (MQLONG) - input

Queue manager flags indicating properties of the queue manager that hosts the instance of the destination queue described by the MQWDR structure. The following flags are defined:

MQQMF_REPOSITORY_Q_MGR

Destination is a full repository queue manager.

MQQMF_CLUSSDR_USER_DEFINED

Cluster-sender channel was defined manually.

MQQMF_CLUSSDR_AUTO_DEFINED

Cluster-sender channel was defined automatically.

MQQMF_AVAILABLE

Destination queue manager is available to receive messages.

Other values

Other flags in the field might be set by the queue manager for internal purposes.

QMgrIdentifier (MQCHAR48) - input

The queue manager identifier is a unique identifier for the queue manager that hosts the instance of the destination queue described by the MQWDR structure.

- The identifier is generated by the queue manager.
- The length of QMgrIdentifier is MQ_Q_MGR_IDENTIFIER_LENGTH.

QMgrName (MQCHAR48) - input

The name of the queue manager that hosts the instance of the destination queue described by the MQWDR structure.

- QMgrName can be the name of the local queue manager, as well another queue manager in the cluster.
- The length of QMgrName is MQ_Q_MGR_NAME_LENGTH.

ClusterRecOffset (MQLONG) - input

The logical offset of the first MQWCR structure that belongs to the MQWDR structure.

- For static caches, ClusterRecOffset is the offset of the first MQWCR structure that belongs to the MQWDR structure.
- The offset is measured in bytes from the start of the MQWDR structure.
- Do not use the logical offset for pointer arithmetic with dynamic caches. To obtain the address of the next record, the MQXCLWLN call must be used.

ChannelState (MQLONG) - input

The state of the channel that links the local queue manager to the queue manager identified by the MQWDR structure. The following values are possible:

MQCHS_BINDING

Channel is negotiating with the partner.

MQCHS_INACTIVE

Channel is not active.

MQCHS_INITIALIZING

Channel is initializing.

MQCHS_PAUSED

Channel has paused.

MQCHS_REQUESTING

Requester channel is requesting connection.

MQCHS_RETRYING

Channel is reattempting to establish connection.

MQCHS_RUNNING

Channel is transferring or waiting for messages.

MQCHS_STARTING

Channel is waiting to become active.

MQCHS_STOPPING

Channel is stopping.

MQCHS_STOPPED

Channel has stopped.

ChannelDefOffset (MQLONG) - input

The logical offset of the channel definition (MQCD) for the channel that links the local queue manager to the queue manager identified by the MQWDR structure.

- ChannelDefOffset is like ClusterRecOffset
- The logical offset cannot be used in pointer arithmetic. To obtain the address of the next record, the MQXCLWLN call must be used.

DestSeqFactor (MQLONG) - input

The destination sequence factor that allows a choice of the channel based on weight.

- DestSeqFactor is used before the queue manager changes it.
- The workload manager increases DestSeqFactor in a way that ensures messages are distributed down channels according to their weight.

DestSeqNumber (MQLONG) - input

The cluster channel destination value before the queue manager changes it.

- The workload manager increases DestSeqNumber every time a message is put down that channel.
- Workload exits can use DestSeqNumber to decide which channel to put a message down.

Related reference

[Initial values and language declarations for MQWDR](#)

Initial values and C and High Level Assembler Language declarations for MQWDR - Cluster workload destination record.

Initial values and language declarations for MQWDR

Initial values and C and High Level Assembler Language declarations for MQWDR - Cluster workload destination record.

Table 650. Initial values of fields in MQWDR		
Field name	Name of constant	Value of constant
StrucId	MQWDR_STRUC_ID	'WDR↵'
Version	MQWDR_VERSION_1	1
StrucLength	MQWDR_CURRENT_LENGTH ³	136
QMgrFlags	MQWDR_NONE	0
QMgrIdentifier	None	" "
QMgrName	None	" "
ClusterRecOffset	None	0
ChannelState	None	0
ChannelDefOffset	None	0
DestSeqNumber	None	0
DestSeqFactor	None	0
Notes:		
1. The symbol ↵ represents a single blank character.		
2. In the C programming language, the macro variable MQWDR_DEFAULT contains the default values. Use it in the following way to provide initial values for the fields in the structure:		
<pre>MQWDR MyWDR = {MQWDR_DEFAULT};</pre>		
3. The initial values intentionally set the length of the structure to the length of the current version, and not version 1 of the structure.		

High Level Assembler

MQWDR	DSECT	
MQWDR_STRUCID	DS CL4	Structure identifier
MQWDR_VERSION	DS F	Structure version number
MQWDR_STRUCLNGTH	DS F	Length of MQWDR structure

MQWDR_QMGRFLAGS	DS	F	Queue-manager flags
MQWDR_QMGRIDENTIFIER	DS	CL48	Queue-manager identifier
MQWDR_QMGRNAME	DS	CL48	Queue-manager name
MQWDR_CLUSTERRECOFFSET	DS	F	Offset of first cluster
*			record
MQWDR_CHANNELSTATE	DS	F	Channel state
MQWDR_CHANNELDEFOFFSET	DS	F	Offset of channel definition
*			structure
MQWDR_LENGTH	EQU	*-MQWDR	Length of structure
	ORG	MQWDR	
MQWDR_AREA	DS	CL(MQWDR_LENGTH)	

C declaration

```
typedef struct tagMQWDR {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     StrucLength;       /* Length of MQWDR structure */
    MQLONG     QMgrFlags;        /* Queue-manager flags */
    MQCHAR48   QMgrIdentifier;    /* Queue-manager identifier */
    MQCHAR48   QMgrName;         /* Queue-manager name */
    MQLONG     ClusterRecOffset; /* Offset of first cluster record */
    MQLONG     ChannelState;      /* Channel state */
    MQLONG     ChannelDefOffset; /* Offset of channel definition structure */
    /* Ver:1 */
    MQLONG     DestSeqNumber;     /* Cluster channel destination sequence number */
    MQINT64    DestSeqFactor;     /* Cluster channel factor sequence number */
    /* Ver:2 */
};
```

Related reference

[Fields in MQWDR - Cluster workload destination record structure](#)

Description of the parameters in the MQWDR - Cluster workload destination record structure.

MQWQR - Cluster workload queue record structure

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

Table 651. Fields in MQWQR		
Field	Description	Page
<i>StrucId</i>	Structure identifier	StrucId
<i>Version</i>	Structure version number	Version
<i>StrucLength</i>	Length of MQWQR structure	StrucLength
<i>QFlags</i>	Queue flags	QFlags
<i>QName</i>	Queue name	QName
<i>QMgrIdentifier</i>	Queue-manager identifier	QMgrIdentifier
<i>ClusterRecOffset</i>	Offset of first cluster record (MQWCR)	ClusterRecOffset
<i>QType</i>	Queue type	QType
<i>QDesc</i>	Queue description	QDesc
<i>DefBind</i>	Default binding	DefBind
<i>DefPersistence</i>	Default message persistence	DefPersistence
<i>DefPriority</i>	Default message priority	DefPriority
<i>InhibitPut</i>	Whether put operations on the queue are allowed	InhibitPut
Note: The remaining fields are ignored if Version is less than MQWQR_VERSION_2.		
<i>CWLQueuePriority</i>	A value 0 - 9 representing the priority of the queue	CLWLQueuePriority

Table 651. Fields in MQWQR (continued)		
Field	Description	Page
<i>CLWLQueueRank</i>	A value 0 - 9 representing the rank of the queue	CLWLQueueRank
Note: The remaining fields are ignored if Version is less than MQWQR_VERSION_3.		
<i>DefPutResponse</i>	Default put response	DefPutResponse

The cluster workload queue record structure contains information relating to one of the possible destinations for the message. There is one cluster workload queue record structure for each instance of the destination queue.

The cluster workload queue record structure is supported in all environments.

Additionally, the MQWQR1 and MQWQR2 structures are available for backwards compatibility.

Related reference

[MQ_CLUSTER_WORKLOAD_EXIT](#) - Call description

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

[MQXCLWLN](#) - Navigate Cluster workload records

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

[MQWXP](#) - Cluster workload exit parameter structure

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

[MQWDR](#) - Cluster workload destination record structure

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

[MQWCR](#) - Cluster workload cluster record structure

The following table summarizes the fields in the MQWCR cluster workload record structure.

Fields in MQWQR - Cluster workload queue record structure

Description of the fields in the MQWQR - Cluster workload queue record structure.

StrucId (MQCHAR4) - input

The structure identifier for the cluster workload queue record structure.

- The StrucId value is MQWQR_STRUC_ID.
- For the C programming language, the constant MQWQR_STRUC_ID_ARRAY is also defined. It has the same value as MQWQR_STRUC_ID. It is an array of characters instead of a string.

Version (MQLONG) - input

The structure version number. Version takes one of the following values:

MQWQR_VERSION_1

Version-1 cluster workload queue record.

MQWQR_VERSION_2

Version-2 cluster workload queue record.

MQWQR_VERSION_3

Version-3 cluster workload queue record.

MQWQR_CURRENT_VERSION

Current version of cluster workload queue record.

StrucLength (MQLONG) - input

The length of MQWQR structure. StrucLength takes one of the following values:

MQWQR_LENGTH_1

Length of version-1 cluster workload queue record.

MQWQR_LENGTH_2

Length of version-2 cluster workload queue record.

MQWQR_LENGTH_3

Length of version-3 cluster workload queue record.

MQWQR_CURRENT_LENGTH

Length of current version of cluster workload queue record.

QFlags (MQLONG) - input

The queue flags indicate properties of the queue. The following flags are defined:

MQQF_LOCAL_Q

Destination is a local queue.

MQQF_CLWL_USEQ_ANY

Allow use of local and remote queues in puts.

MQQF_CLWL_USEQ_LOCAL

Allow only local queue puts.

Other values

Other flags in the field might be set by the queue manager for internal purposes.

QName (MQCHAR48) - input

The name of the queue that is one of the possible destinations of the message.

- The length of QName is MQ_Q_NAME_LENGTH.

QMgrIdentifier (MQCHAR48) - input

The queue manager identifier is a unique identifier for the queue manager that hosts the instance of the queue described by the MQWQR structure.

- The identifier is generated by the queue manager.
- The length of QMgrIdentifier is MQ_Q_MGR_IDENTIFIER_LENGTH.

ClusterRecOffset (MQLONG) - input

The logical offset of the first MQWCR structure that belongs to the MQWQR structure.

- For static caches, ClusterRecOffset is the offset of the first MQWCR structure that belongs to the MQWQR structure.
- The offset is measured in bytes from the start of the MQWQR structure.
- Do not use the logical offset for pointer arithmetic with dynamic caches. To obtain the address of the next record, the MQXCLWLN call must be used.

QType (MQLONG) - input

The queue type of the destination queue. The following values are possible:

MQCQT_LOCAL_Q

Local queue.

MQCQT_ALIAS_Q

Alias queue.

MQCQT_REMOTE_Q

Remote queue.

MQCQT_Q_MGR_ALIAS

Queue-manager alias.

QDesc (MQCHAR64) - input

The queue description queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure.

- The length of QDesc is MQ_Q_DESC_LENGTH.

DefBind (MQLONG) - input

The default binding queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure. Either MQBND_BIND_ON_OPEN or

MQBND_BIND_ON_GROUP must be specified when using groups with clusters. The following values are possible:

MQBND_BIND_ON_OPEN

Binding fixed by MQOPEN call.

MQBND_BIND_NOT_FIXED

Binding not fixed.

MQBND_BIND_ON_GROUP

Allows an application to request that a group of messages are all allocated to the same destination instance.

DefPersistence (MQLONG) - input

The default message persistence queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure. The following values are possible:

MQPER_PERSISTENT

Message is persistent.

MQPER_NOT_PERSISTENT

Message is not persistent.

DefPriority (MQLONG) - input

The default message priority queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure. The priority range is 0 - MaxPriority.

- 0 is the lowest priority.
- MaxPriority is the queue manager attribute of the queue manager that hosts this instance of the destination queue.

InhibitPut (MQLONG) - input

The put inhibited queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure. The following values are possible:

MQQA_PUT_INHIBITED

Put operations are inhibited.

MQQA_PUT_ALLOWED

Put operations are allowed.

CLWLQueuePriority (MQLONG) - input

The cluster workload queue priority attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure.

CLWLQueueRank (MQLONG) - input

The cluster workload queue rank defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure.

DefPutResponse (MQLONG) - input

The default put response queue attribute defined on the queue manager that hosts the instance of the destination queue described by the MQWQR structure. The following values are possible:

MQPRT_SYNC_RESPONSE

Synchronous response to MQPUT or MQPUT1 calls.

MQPRT_ASYNC_RESPONSE

Asynchronous response to MQPUT or MQPUT1 calls.

Related reference

[Initial values and language declarations for MQWQR](#)

Initial values and C and High Level Assembler Language declarations for MQWQR - Cluster workload queue record.

Initial values and language declarations for MQWQR

Initial values and C and High Level Assembler Language declarations for MQWQR - Cluster workload queue record.

Table 652. Initial values of fields in MQWQR		
Field name	Name of constant	Value of constant
<i>StrucId</i>	MQWQR_STRUC_ID_ARRAY	'WQR↵'
<i>Version</i>	MQWQR_VERSION_1	1
<i>StrucLength</i>	MQWQR_CURRENT_LENGTH ³	212
<i>QFlags</i>	None	0
<i>QName</i>	None	" "
<i>QMgrIdentifier</i>	None	" "
<i>ClusterRecOffset</i>	None	0
<i>QType</i>	None	0
<i>QDesc</i>	None	" "
<i>DefBind</i>	None	0
<i>DefPersistence</i>	None	0
<i>DefPriority</i>	None	0
<i>InhibitPut</i>	None	0
<i>CLWLQueuePriority</i>	None	0
<i>CLWLQueueRank</i>	None	0
<i>DefPutResponse</i>	None	1
Notes: <ol style="list-style-type: none"> 1. The symbol ↵ represents a single blank character. 2. In the C programming language, the macro variable MQWQR_DEFAULT contains the default values. Use it in the following way to provide initial values for the fields in the structure: <pre>MQWQR MyWQR = {MQWQR_DEFAULT};</pre> 3. The initial values intentionally set the length of the structure to the length of the current version, and not version 1 of the structure. 		

C declaration

```
typedef struct tagMQWQR {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;           /* Structure version number */
    MQLONG     StrucLength;       /* Length of MQWQR structure */
    MQLONG     QFlags;            /* Queue flags */
    MQCHAR48   QName;            /* Queue name */
    MQCHAR48   QMgrIdentifier;     /* Queue-manager identifier */
    MQLONG     ClusterRecOffset;  /* Offset of first cluster record */
    MQLONG     QType;            /* Queue type */
    MQCHAR64   QDesc;            /* Queue description */
    MQLONG     DefBind;          /* Default binding */
    MQLONG     DefPersistence;    /* Default message persistence */
    MQLONG     DefPriority;       /* Default message priority */
}
```

```

    MQLONG    InhibitPut;          /* Whether put operations on the queue
                                   are allowed */
    /* version 2 */
    MQLONG    CLWLQueuePriority; /* Queue priority */
    MQLONG    CLWLQueueRank;     /* Queue rank */
    /* version 3 */
    MQLONG    DefPutResponse;     /* Default put response */
};

```

High Level Assembler

```

MQWQR                                DSECT
MQWQR_STRUCID                       DS    CL4      Structure identifier
MQWQR_VERSION                       DS    F         Structure version number
MQWQR_STRUCLNGTH                    DS    F         Length of MQWQR structure
MQWQR_QFLAGS                        DS    F         Queue flags
MQWQR_QNAME                         DS    CL48      Queue name
MQWQR_QMGRIDENTIFIER                DS    CL48      Queue-manager identifier
MQWQR_CLUSTERRECOFFSET              DS    F         Offset of first cluster
*                                   record
MQWQR_QTYPE                         DS    F         Queue type
MQWQR_QDESC                         DS    CL64      Queue description
MQWQR_DEFBIND                       DS    F         Default binding
MQWQR_DEFPERSISTENCE                DS    F         Default message persistence
MQWQR_DEFPRIORITY                   DS    F         Default message priority
MQWQR_INHIBITPUT                    DS    F         Whether put operations on
*                                   the queue are allowed
MQWQR_DEFPUTRESPONSE                DS    F         Default put response
MQWQR_LENGTH                        EQU    *-MQWQR   Length of structure
                                   ORG    MQWQR
MQWQR_AREA                          DS    CL(MQWQR_LENGTH)

```

Related reference

[Fields in MQWQR - Cluster workload queue record structure](#)

Description of the fields in the MQWQR - Cluster workload queue record structure.

MQWCR - Cluster workload cluster record structure

The following table summarizes the fields in the MQWCR cluster workload record structure.

Table 653. Fields in MQWCR.		
There are three columns in this table. The first column lists the field names, the second column provides a description of each field and the third column provides a link to further information about each field.		
Field	Description	Page
<i>ClusterName</i>	Name of cluster	ClusterName
<i>ClusterRecOffset</i>	Offset of next cluster record (MQWCR)	ClusterRecOffset
<i>ClusterFlags</i>	Cluster flags	ClusterFlags

The cluster workload cluster record structure contains information about a cluster. For each cluster the destination queue belongs to, there is one cluster workload cluster record structure.

The cluster workload cluster record structure is supported in all environments.

Related reference

[MQ_CLUSTER_WORKLOAD_EXIT - Call description](#)

The cluster workload exit is called by the queue manager to route a message to an available queue manager.

[MQXCLWLN - Navigate Cluster workload records](#)

The MQXCLWLN call is used to navigate through the chains of MQWDR, MQWQR, and MQWCR records stored in the cluster cache.

[MQWXP - Cluster workload exit parameter structure](#)

The following table summarizes the fields in the MQWXP - Cluster workload exit parameter structure.

MQWDR - Cluster workload destination record structure

The following table summarizes the fields in the MQWDR - Cluster workload destination record structure.

MQWQR - Cluster workload queue record structure

The following table summarizes the fields in the MQWQR - Cluster workload queue record structure.

Fields in the MQWCR - Cluster workload cluster record structure.

Description of the fields in the MQWCR - Cluster workload cluster record structure.

ClusterName (MQCHAR48) - input

The name of a cluster to which the instance of the destination queue that owns the MQWCR structure belongs. The destination queue instance is described by an MQWDR structure.

- The length of ClusterName is MQ_CLUSTER_NAME_LENGTH.

ClusterRecOffset (MQLONG) - input

The logical offset of the next MQWCR structure.

- If there are no more MQWCR structures, ClusterRecOffset is zero.
- The offset is measured in bytes from the start of the MQWCR structure.

ClusterFlags (MQLONG) - input

The cluster flags indicate properties of the queue manager identified by the MQWCR structure. The following flags are defined:

MQQMF_REPOSITORY_Q_MGR

Destination is a full repository queue manager.

MQQMF_CLUSSDR_USER_DEFINED

Cluster-sender channel was defined manually.

MQQMF_CLUSSDR_AUTO_DEFINED

Cluster-sender channel was defined automatically.

MQQMF_AVAILABLE

Destination queue manager is available to receive messages.

Other values

Other flags in the field might be set by the queue manager for internal purposes.

Related reference

Initial values and language declarations for MQWCR

Initial values and C and High Level Assembler Language declarations for MQWCR - Cluster workload cluster record structure.

Initial values and language declarations for MQWCR

Initial values and C and High Level Assembler Language declarations for MQWCR - Cluster workload cluster record structure.

Table 654. Initial values of fields in MQWCR		
Field name	Name of constant	Value of constant
ClusterName	None	" "
ClusterRecOffset	None	0
ClusterFlags	None	0

C declaration

```
typedef struct tagMQWCR {  
    MQCHAR48  ClusterName;      /* Cluster name */  
    MQLONG    ClusterRecOffset; /* Offset of next cluster record */  
}
```

```

    MQLONG    ClusterFlags;    /* Cluster flags */
};

```

High Level Assembler

MQWCR	DSECT	
MQWCR_CLUSTERNAME	DS	CL48 Cluster name
MQWCR_CLUSTERRECOFFSET	DS	F Offset of next cluster record
*MQWCR_CLUSTERFLAGS	DS	F Cluster flags
MQWCR_LENGTH	EQU	*-MQWCR Length of structure
	ORG	MQWCR
MQWCR_AREA	DS	CL(MQWCR_LENGTH)

Related reference

[Fields in the MQWCR - Cluster workload cluster record structure.](#)

Description of the fields in the MQWCR - Cluster workload cluster record structure.

API exit reference

This section provides reference information mainly of interest to a programmer writing API exits.

General usage notes

notes:

1. All exit functions can issue the MQXEP call; this call is designed specifically for use from API exit functions.
2. The MQ_INIT_EXIT function cannot issue any MQ calls other than MQXEP.
3. You cannot issue the MQDISC call for the current connection.
4. If an exit function issues the MQCONN call, or the MQCONN call with the MQCNO_HANDLE_SHARE_NONE option, the call completes with reason code MQRC_ALREADY_CONNECTED, and the handle returned is the same as the one passed to the exit as a parameter.
5. In general when an API exit function issues an MQI call, API exits are not be called recursively. However, if an exit function issues the MQCONN call with the MQCNO_HANDLE_SHARE_BLOCK or MQCNO_HANDLE_SHARE_NO_BLOCK options, the call returns a new shared handle. This provides the exit suite with a connection handle of its own, and hence a unit of work that is independent of the application's unit of work. The exit suite can use this handle to put and get messages within its own unit of work, and commit or back out that unit of work; all of this can be done without affecting the application's unit of work in any way.

Because the exit function is using a connection handle that is different from the handle being used by the application, MQ calls issued by the exit function result in the relevant API exit functions being invoked. Exit functions can therefore be invoked recursively. Note that both the *ExitUserArea* field in MQAXP and the exit chain area have connection-handle scope. Consequently, an exit function cannot use those areas to signal to another instance of itself invoked recursively that it is already active.

6. Exit functions can also put and get messages within the application's unit of work. When the application commits or backs out the unit of work, all messages within the unit of work are committed or backed out together, regardless of who placed them in the unit of work (application or exit function). However, the exit can cause the application to exceed system limits sooner than would otherwise be the case (for example, by exceeding the maximum number of uncommitted messages in a unit of work).

When an exit function uses the application's unit of work in this way, the exit function should usually avoid issuing the MQCMIT call, as this commits the application's unit of work and might impair the correct functioning of the application. However, the exit function might sometimes need to issue the MQBACK call, if the exit function encounters a serious error that prevents the unit of work being committed (for example, an error putting a message as part of the application's unit of work). When

MQBACK is called, take care to ensure that the application unit of work boundaries are not changed. In this situation the exit function must set the appropriate values to ensure that completion code MQCC_WARNING and reason code MQRC_BACKED_OUT are returned to the application, so that the application can detect the fact that the unit of work has been backed out.

If an exit function uses the application's connection handle to issue MQ calls, those calls do not themselves result in further invocations of API exit functions.

7. If an MQXR_BEFORE exit function terminates abnormally, the queue manager might be able to recover from the failure. If it can, the queue manager continues processing as though the exit function had returned MQXCC_FAILED. If the queue manager cannot recover, the application is terminated.
8. If an MQXR_AFTER exit function terminates abnormally, the queue manager might be able to recover from the failure. If it can, the queue manager continues processing as though the exit function had returned MQXCC_FAILED. If the queue manager cannot recover, the application is terminated. Be aware that in the latter case, messages retrieved outside a unit of work are lost (this is the same situation as the application failing immediately after removing a message from the queue).
9. The MCA process performs a two phase commit.

If an API exit intercepts an MQCMIT from a prepared MCA process and attempts to perform an action within the unit of work, then the action will fail with reason code MQRC_UOW_NOT_AVAILABLE.
10. For a multi-installation environment, the only way to have an exit that works with both Websphere MQ Version 7.0 and Version 7.1 is to write the exit in a way which links at Version 7.0 with mqm.Lib and, for non-primary or relocated exits, to ensure that the application finds the correct mqm.Lib for the installation with which the queue manager is currently associated, prior to the application launch. (For example, run the **setmqenv -m QM** command before launching the application, even if the queue manager is owned by a Version 7.0 installation.)
11. Where multiple installations of IBM MQ are available, use the exits written for an earlier version of IBM MQ, as new functionality added in the later version might not work with earlier versions. For more information about changes between releases, see [What's changed in IBM MQ 8.0](#).

IBM MQ API exit parameter structure (MQAXP)

The MQAXP structure, an external control block, is used as an input or output parameter to the API exit. This topic also gives information about how queue managers process exit functions.

MQAXP has the following C declaration:

```
typedef struct tagMQAXP {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     ExitId;           /* Exit Identifier */
    MQLONG     ExitReason;       /* Exit invocation reason */
    MQLONG     ExitResponse;     /* Response code from exit */
    MQLONG     ExitResponse2;    /* Secondary response code from exit */
    MQLONG     Feedback;        /* Feedback code from exit */
    MQLONG     APICallerType;    /* MQSeries API caller type */
    MQBYTE16   ExitUserArea;     /* User area for use by exit */
    MQCHAR32   ExitData;        /* Exit data area */
    MQCHAR48   ExitInfoName;    /* Exit information name */
    MQBYTE48   ExitPDArea;      /* Problem determination area */
    MQCHAR48   QMgrName;        /* Name of local queue manager */
    PMQACH     ExitChainAreaPtr; /* Inter exit communication area */
    MQHCONFIG  Hconfig;         /* Configuration handle */
    MQLONG     Function;        /* Function Identifier */
    /* Ver:1 */
    MQHMSG     ExitMsgHandle     /* Exit message handle */
    /* Ver:2 */
};
```

The following parameter list is passed when functions in an API exit are invoked:

StrucId (MQCHAR4) - input

The exit parameter structure identifier, with a value of:

MQAXP_STRUC_ID.

The exit handler sets this field on entry to each exit function.

Version (MQLONG) - input

The structure version number, with a value of:

MQAXP_VERSION_1

Version 1 API exit parameter structure.

MQAXP_VERSION_2

Version 2 API exit parameter structure.

MQAXP_CURRENT_VERSION

Current version number for the API exit parameter structure.

The exit handler sets this field on entry to each exit function.

ExitId (MQLONG) - input

The exit identifier, set on entry to the exit routine, indicating the type of exit:

MQXT_API_EXIT

API exit.

ExitReason (MQLONG) - input

The reason for invoking the exit, set on entry to each exit function:

MQXR_CONNECTION

The exit is being invoked to initialize itself before an MQCONN or MQCONNEX call, or to end itself after an MQDISC call.

MQXR_BEFORE

The exit is being invoked before executing an API call, or before converting data on an MQGET.

MQXR_AFTER

The exit is being invoked after executing an API call.

ExitResponse (MQLONG) - output

The response from the exit, initialized on entry to each exit function to:

MQXCC_OK

Continue normally.

This field must be set by the exit function, to communicate to the queue manager the result of executing the exit function. The value must be one of the following:

MQXCC_OK

The exit function completed successfully. Continue normally.

This value can be set by all MQXR_* exit functions. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

MQXCC_FAILED

The exit function failed because of an error.

This value can be set by all MQXR_* exit functions. The queue manager sets CompCode to MQCC_FAILED, and Reason to:

- MQRC_API_EXIT_INIT_ERROR if the function is MQ_INIT_EXIT
- MQRC_API_EXIT_TERM_ERROR if the function is MQ_TERM_EXIT
- MQRC_API_EXIT_ERROR for all other exit functions

The values set can be altered by an exit function later in the chain.

ExitResponse2 is ignored; the queue manager continues processing as though MQXR2_SUPPRESS_CHAIN had been returned.

MQXCC_SUPPRESS_FUNCTION

Suppress IBM MQ API function.

This value can be set only by an MQXR_BEFORE exit function. It bypasses the API call. If it is returned by the MQ_DATA_CONV_ON_GET_EXIT, data conversion is bypassed. The queue manager sets CompCode to MQCC_FAILED, and Reason to MQRC_SUPPRESSED_BY_EXIT, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

If this value is set by an MQXR_AFTER or MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

MQXCC_SKIP_FUNCTION

Skip IBM MQ API function.

This value can be set only by an MQXR_BEFORE exit function. It bypasses the API call. If it is returned by the MQ_DATA_CONV_ON_GET_EXIT, data conversion is bypassed. The exit function must set CompCode and Reason to the values to be returned to the application, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

If this value is set by an MQXR_AFTER or MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

MQXCC_SUPPRESS_EXIT

Suppress all exit functions belonging to the set of exits.

This value can be set only by the MQXR_BEFORE and MQXR_AFTER exit functions. It bypasses *all* subsequent invocations of exit functions belonging to this set of exits for this logical connection. This bypassing continues until the logical disconnect request occurs, when MQ_TERM_EXIT function is invoked with an ExitReason of MQXR_CONNECTION.

The exit function must set CompCode and Reason to the values to be returned to the application, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is ignored.

If this value is set by an MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

For information about the interaction between ExitResponse and ExitResponse2, and its effect on exit processing, see [“How queue managers process exit functions” on page 1571](#).

ExitResponse2 (MQLONG) - output

This is a secondary exit response code that qualifies the primary exit response code for MQXR_BEFORE exit functions. It is initialized to:

```
MQXR2_DEFAULT_CONTINUATION
```

on entry to an IBM MQ API call exit function. It can then be set to one of the values:

MQXR2_DEFAULT_CONTINUATION

Whether to continue with the next exit in the chain, depending on the value of ExitResponse.

If ExitResponse is MQXCC_SUPPRESS_FUNCTION or MQXCC_SKIP_FUNCTION, bypass exit functions later in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain. Invoke exit functions in the MQXR_AFTER chain that match exit functions earlier in the MQXR_BEFORE chain.

Otherwise, invoke the next exit in the chain.

MQXR2_SUPPRESS_CHAIN

Suppress the chain.

Bypass exit functions later in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain for this API call invocation. Invoke exit functions in the MQXR_AFTER chain that match exit functions earlier in the MQXR_BEFORE chain.

MQXR2_CONTINUE_CHAIN

Continue with the next exit in the chain.

For information about the interaction between ExitResponse and ExitResponse2, and its effect on exit processing, see [“How queue managers process exit functions” on page 1571](#).

Feedback (MQLONG) - input/output

Communicate feedback codes between exit function invocations. This is initialized to:

```
MQFB_NONE (0)
```

before invoking the first function of the first exit in a chain.

Exits can set this field to any value, including any valid MQFB_* or MQRC_* value. Exits can also set this field to a user-defined feedback value in the range MQFB_APPL_FIRST to MQFB_APPL_LAST.

APICallerType (MQLONG) - input

The API caller type, indicating whether the IBM MQ API caller is external or internal to the queue manager: MQXACT_EXTERNAL or MQXACT_INTERNAL.

ExitUserArea (MQBYTE16) - input/output

A user area, available to all the exits associated with a particular ExitInfoObject. It is initialized to MQXUA_NONE (binary zeros for the length of the ExitUserArea) before invoking the first exit function (MQ_INIT_EXIT) for the hconn. From then on, any changes made to this field by an exit function are preserved across invocations of functions of the same exit.

This field is aligned to a multiple of 4 MQLONGs.

Exits can also anchor any storage that they allocate from this area.

For each hconn, each exit in a chain of exits has a different ExitUserArea. The ExitUserArea cannot be shared by exits in a chain, and the contents of the ExitUserArea for one exit are not available to another exit in a chain.

For C programs, the constant MQXUA_NONE_ARRAY is also defined with the same value as MQXUA_NONE, but as an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH.

ExitData (MQCHAR32) - input

Exit data, set on input to each exit function to the 32 characters of exit-specific data that is provided in the exit. If you define no value in the exit this field is all blanks.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

ExitInfoName (MQCHAR48) - input

The exit information name, set on input to each exit function to the ApiExit_name specified in the exit definitions in the stanzas.

ExitPDArea (MQBYTE48) - input/output

A problem determination area, initialized to MQXPDA_NONE (binary zeros for the length of the field) for each invocation of an exit function.

For C programs, the constant MQXPDA_NONE_ARRAY is also defined with the same value as MQXPDA_NONE, but as an array of characters instead of a string.

The exit handler always writes this area to the IBM MQ trace at the end of an exit, even when the function is successful.

The length of this field is given by MQ_EXIT_PD_AREA_LENGTH.

QMgrName (MQCHAR48) - input

The name of the queue manager the application is connected to, that has invoked an exit as a result of processing an IBM MQ API call.

If the name of a queue manager supplied on an MQCONN or MQCONNx calls is blank, this field is still set to the name of the queue manager to which the application is connected, whether the application is server or client.

The exit handler sets this field on entry to each exit function.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH.

ExitChainAreaPtr (PMQACH) - input/output

This is used to communicate data across invocations of different exits in a chain. It is set to a NULL pointer before invoking the first function (MQ_INIT_EXIT with ExitReason MQXR_CONNECTION) of the first exit in a chain of exits. The value returned by the exit on one invocation is passed on to the next invocation.

Refer to [“The exit chain area and exit chain area header \(MQACH\)” on page 1575](#) for more details about how to use the exit chain area.

Hconfig (MQHCONFIG) - input

The configuration handle, representing the set of functions being initialized. This value is generated by the queue manager on the MQ_INIT_EXIT function, and is later passed to the API exit function. It is set on entry to each exit function.

You can use Hconfig as a pointer to the MQIEP structure to make MQI and DCI calls. You must check that the first 4 bytes of HConfig match the StrucId of the MQIEP structure before using the HConfig parameter as a pointer to the MQIEP structure.

Function (MQLONG) - input

The function identifier, valid values for which are the MQXF_* constants described in [“External constants” on page 1576](#).

The exit handler sets this field to the correct value, on entry to each exit function, depending on the IBM MQ API call that resulted in the exit being invoked.

ExitMsgHandle (MQHMSG) - input/output

When Function is MQXF_GET and ExitReason is MQXR_AFTER, a valid message handle is returned in this field allowing the API exit access to the message descriptor fields and any other properties matching the ExitProperties string specified in the MQXEPO structure when registering the API exit.

Any non-message descriptor properties that are returned in the ExitMsgHandle will not be available from the MsgHandle in the MQGMO structure if one was specified, or in the message data.

When Function is MQXF_GET and ExitReason is MQXR_BEFORE, if the exit program sets this field to MQHM_NONE then it will suppress the populating of the ExitMsgHandle properties.

This field is not set if Version is less than MQAXP_VERSION_2.

How queue managers process exit functions

The processing performed by the queue manager on return from an exit function depends on both ExitResponse and ExitResponse2.

[Table 655 on page 1572](#) summarizes the possible combinations and their effects for an MQXR_BEFORE exit function, showing:

- Who sets the CompCode and Reason parameters of the API call
- Whether the remaining exit functions in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain are invoked
- Whether the API call is invoked

For an MQXR_AFTER exit function:

- CompCode and Reason are set in the same way as MQXR_BEFORE
- ExitResponse2 is ignored (the remaining exit functions in the MQXR_AFTER chain are always invoked)
- MQXCC_SUPPRESS_FUNCTION and MQXCC_SKIP_FUNCTION are not valid

For an MQXR_CONNECTION exit function:

- CompCode and Reason are set in the same way as MQXR_BEFORE
- ExitResponse2 is ignored
- MQXCC_SUPPRESS_FUNCTION, MQXCC_SKIP_FUNCTION, MQXCC_SUPPRESS_EXIT are not valid

In all cases, where an exit or the queue manager sets `CompCode` and `Reason`, the values set can be changed by an exit invoked later, or by the API call (if the API call is invoked later).

Table 655. MQXR_BEFORE exit processing			
Value of ExitResponse	CompCode and Reason set by	Value of ExitResponse2 (default continuation) Chain	Value of ExitResponse2 (default continuation) API
MQXCC_OK	exit	Y	Y
MQXCC_SUPPRESS_EXIT	exit	Y	Y
MQXCC_SUPPRESS_FUNCTION	queue manager	N	N
MQXCC_SKIP_FUNCTION	exit	N	N
MQXCC_FAILED	queue manager	N	N

How clients process exit functions

In general, clients process exit functions in the same way that server applications do, and the *QMGrName* attribute in this structure applies whether the function is on a server or a client.

However, the client has no concept of the *mqs.ini* file, so the *ApiExitCommon* and *APIExitTemplate* stanzas do not apply. Only the *ApiExitLocal* stanza applies, and this stanza is configured in the *mqclient.ini* file.

IBM MQ API exit context structure (MQAXC)

The MQAXC structure, an external control block, is used as an input parameter to an API exit.

MQAXC has the following C declaration:

```
typedef struct tagMQAXC {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     Environment;      /* Environment */
    MQCHAR12   UserId;           /* UserId associated with appl */
    MQBYTE40   SecurityId;       /* Extension to UserId running appl */
    MQCHAR264  ConnectionName;   /* Connection name */
    MQLONG     LongMCAUserIdLength; /* long MCA user identifier length */
    MQLONG     LongRemoteUserIdLength; /* long remote user identifier length */
    MQPTR      LongMCAUserIdPtr; /* long MCA user identifier address */
    MQPTR      LongRemoteUserIdPtr; /* long remote user identifier address */
    MQCHAR28   ApplName;         /* Application name */
    MQLONG     ApplType;         /* Application type */
    MQPID      ProcessId;        /* Process identifier */
    MQTID      ThreadId;         /* Thread identifier */

    /* Ver:1 */
    MQCHAR     ChannelName[20]   /* Channel Name */
    MQBYTE4    Reserved1;        /* Reserved */
    PMQCD      pChannelDefinition; /* Channel Definition pointer */
};
```

The parameters to MQAXC are:

StrucId (MQCHAR4) - input

The exit context structure identifier, with a value of `MQAXC_STRUC_ID`. For C programs, the constant `MQAXC_STRUC_ID_ARRAY` is also defined, with the same value as `MQAXC_STRUC_ID`, but as an array of characters instead of a string.

The exit handler sets this field on entry to each exit function.

Version (MQLONG) - input

The structure version number, with a value of:

MQAXC_VERSION_2

Version number for the exit context structure.

MQAXC_CURRENT_VERSION

Current version number for the exit context structure.

The exit handler sets this field on entry to each exit function.

Environment (MQLONG) - input

The environment from which an IBM MQ API call was issued that resulted in an exit function being driven. Valid values for this field are:

MQXE_OTHER

This value is consistent with invocations an API exit sees if the exit is called from a server application. This means that an API exit runs unchanged on a client and does not see anything different.

If the exit really needs to determine whether it is running on the client, the exit can do so by looking at the *ChannelName* and *ChannelDefinition* fields.

MQXE_MCA

Message channel agent

MQXE_MCA_SVRCONN

A message channel agent acting on behalf of a client

MQXE_COMMAND_SERVER

The command server

MQXE_MQSC

The runmqsc command interpreter

The exit handler sets this field on entry to each exit function.

UserId (MQCHAR12) - input

The user ID associated with the application. In particular, in the case of client connections, this field contains the user ID of the adopted user as opposed to the user ID under which the channel code is running. If a blank user ID flows from the client, then no change is made to the user ID already being used. That is, no new user ID is adopted.

The exit handler sets this field on entry to each exit function. The length of this field is given by MQ_USER_ID_LENGTH.

In the case of a client, this is the user ID sent from the client to the server. Note, that this might not be the effective user ID the client is running against in the queue manager, as there could be an MCAUser or CHLAUTH configuration which changes the user ID.

SecurityId (MQBYTE40) - input

An extension to the user ID running the application. Its length is given by MQ_SECURITY_ID_LENGTH.

In the case of a client, this is the user ID sent from the client to the server. Note, that this might not be the effective user ID the client is running against in the queue manager, as there could be an MCAUser or CHLAUTH configuration which changes the user ID.

ConnectionName (MQCHAR264) - input

The connection name field, set to the address of the client. For example, for TCP/IP, it would be the client IP address.

The length of this field is given by MQ_CONN_NAME_LENGTH.

In the case of a client, this is the partner address of the queue manager.

LongMCAUserIdLength (MQLONG) - input

The length of the long MCA user identifier.

When MCA connects to the queue manager this field is set to the length of the long MCA user identifier (or zero if there is no such identifier).

In the case of a client, this is the client long user identifier.

LongRemoteUserIdLength (MQLONG) - input

The length of the long remote user identifier.

When MCA connects to the queue manager this field is set to the length of the long remote user identifier. Otherwise this field will be set to zero

In the case of a client, set this field to zero.

LongMCAUserIdPtr (MQPTR) - input

Address of long MCA user identifier.

When MCA connects to the queue manager this field is set to the address of the long MCA user identifier (or to a null pointer if there is no such identifier).

In the case of a client, this is the client long user identifier.

LongRemoteUserIdPtr (MQPTR) - input

The address of the long remote user identifier.

When MCA connects to the queue manager this field is set to the address of the long remote user identifier (or to a null pointer if there is no such identifier).

In the case of a client, set this field to zero.

ApplName (MQCHAR28) - input

The name of the application or component that issued the IBM MQ API call.

The rules for generating the ApplName are the same as for generating the default name for an MQPUT.

The value of this field is found by querying the operating system for the program name. Its length is given by MQ_APPL_NAME_LENGTH.

ApplType (MQLONG) - input

The type of application or component that issued the IBM MQ API call.

The value is MQAT_DEFAULT for the platform on which the application is compiled, or it equates to one of the defined MQAT_* values.

The exit handler sets this field on entry to each exit function.

ProcessId (MQPID) - input

The operating system process identifier.

Where applicable, the exit handler sets this field on entry to each exit function.

ThreadId (MQTID) - input

The MQ thread identifier. This is the same identifier used in MQ trace and FFST dumps, but might be different from the operating system thread identifier.

Where applicable, the exit handler sets this field on entry to each exit function.

ChannelName (MQCHAR) - input

The name of the channel, padded with blanks, if applicable and known.

If not applicable, this field is set to NULL characters.

Reserved1 (MQBYTE4) - input

This field is reserved.

ChannelDefinition (PMQCD) - input

A pointer to the channel definition being used, if applicable and known.

If not applicable, this field is set to NULL characters.

Note that the pointer is only completed if the connection is processing on behalf of an IBM MQ channel and that channel definition has been read.

In particular, the channel definition is not given on the server when the first MQCONN call is made for the channel. Furthermore, if the pointer is filled, the structure (and any sub structures) pointed to by the pointer must be treated as read only; any updating of the structure would lead to unpredictable results and is not supported.

In the case of a client, fields other than those with a value specified for a client, contain values that are appropriate for a client application.

The exit chain area and exit chain area header (MQACH)

If required, an exit function can acquire storage for an exit chain area and set the ExitChainAreaPtr in MQAXP to point to this storage.

Exits (either the same or different exit functions) can acquire multiple exit chain areas and link them together. Exit chain areas must only be added or removed from this list while called from the exit handler. This ensures that there are no serialization issues caused by different threads adding or removing areas from the list at the same time.

An exit chain area must start with an MQACH header structure, the C declaration for which is:

```
typedef struct tagMQACH {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQLONG     StrucLength;      /* Length of the MQACH structure */
    MQLONG     ChainAreaLength; /* Exit chain area length */
    MQCHAR48   ExitInfoName;     /* Exit information name */
    PMQACH     NextChainAreaPtr; /* Pointer to next exit chain area */
};
```

The fields in the exit chain area header are:

StrucId (MQCHAR4) - input

The exit chain area structure identifier, with an initial value, defined by MQACH_DEFAULT, of MQACH_STRUC_ID.

For C programs, the constant MQACH_STRUC_ID_ARRAY is also defined; this has the same value as MQACH_STRUC_ID, but as an array of characters instead of a string.

Version (MQLONG) - input

The structure version number, as follows:

MQACH_VERSION_1

The version number for the exit parameter structure.

MQACH_CURRENT_VERSION

The current version number for the exit context structure.

The initial value of this field, defined by MQACH_DEFAULT, is MQACH_CURRENT_VERSION.

Note: If you introduce a new version of this structure, the layout of the existing part does not change. Exit functions must check that the version number is equal to or greater than the lowest version containing the fields that the exit function needs to use.

StrucLength (MQLONG) - input

The length of the MQACH structure. Exits can use this field to determine the start of the exit data, setting it to the length of the structure created by the exit.

The initial value of this field, defined by MQACH_DEFAULT, is MQACH_CURRENT_LENGTH.

ChainAreaLength (MQLONG) - input

The exit chain area length, set to the overall length of the current exit chain area, including the MQACH header.

The initial value of this field, defined by MQACH_DEFAULT, is zero.

ExitInfoName (MQCHAR48) - input

The exit information name.

When an exit creates an MQACH structure, it must initialize this field with its own ExitInfoName, so that later this MQACH structure can be found by either another instance of this exit, or by a cooperating exit.

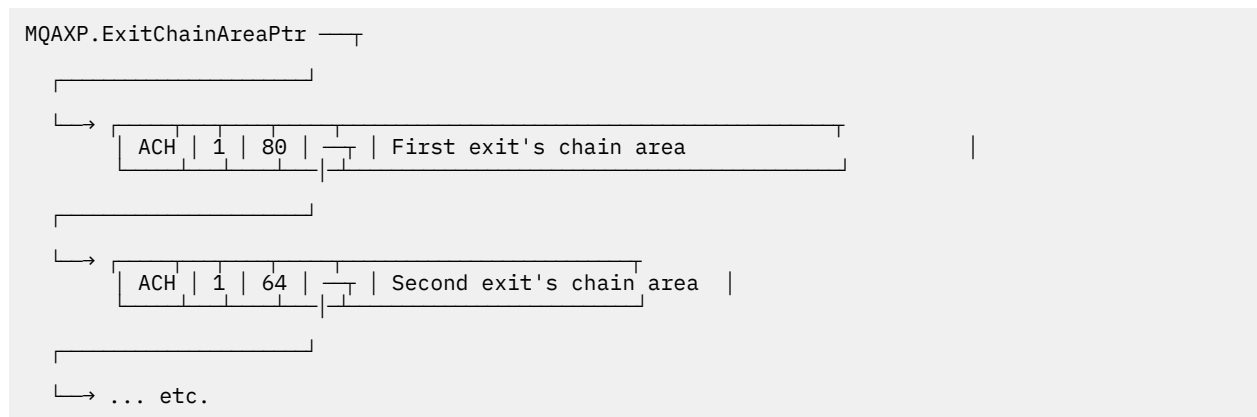
The initial value of this field, defined by MQACH_DEFAULT, is a zero length string ({}).

NextChainAreaPtr (PMQACH) - input

A pointer to the next exit chain area with an initial value, defined by MQACH_DEFAULT, of null pointer (NULL).

Exit functions must release the storage for any exit chain areas that they acquire, and manipulate the chain pointers to remove their exit chain areas from the list.

An exit chain area can be constructed as follows:



External constants

Use this topic as reference information for external constants available for API exits.

The following external constants are available for API exits:

MQXF_* (exit function identifiers)

MQXF_INIT	1	X'00000001'
MQXF_TERM	2	X'00000002'
MQXF_CONN	3	X'00000003'
MQXF_CONNX	4	X'00000004'
MQXF_DISC	5	X'00000005'
MQXF_OPEN	6	X'00000006'
MQXF_CLOSE	7	X'00000007'
MQXF_PUT1	8	X'00000008'
MQXF_PUT	9	X'00000009'
MQXF_GET	10	X'0000000A'
MQXF_DATA_CONV_ON_GET	11	X'0000000B'
MQXF_INQ	12	X'0000000C'
MQXF_SET	13	X'0000000D'
MQXF_BEGIN	14	X'0000000E'
MQXF_CMIT	15	X'0000000F'
MQXF_BACK	16	X'00000010'
MQXF_STAT	18	X'00000012'
MQXF_CB	19	X'00000013'
MQXF_CTL	20	X'00000014'
MQXF_CALLBACK	21	X'00000015'
MQXF_SUB	22	X'00000016'
MQXF_SUBRQ	23	X'00000017'
MQXF_XACLOSE	24	X'00000018'
MQXF_XACOMMIT	25	X'00000019'
MQXF_XACOMplete	26	X'0000001A'
MQXF_XAEND	27	X'0000001B'
MQXF_XAFORGET	28	X'0000001C'
MQXF_XAOPEN	29	X'0000001D'
MQXF_XAPREPARE	30	X'0000001E'
MQXF_XARECOVER	31	X'0000001F'
MQXF_XAROLLBACK	32	X'00000020'
MQXF_XASTART	33	X'00000021'
MQXF_AXREG	34	X'00000022'
MQXF_AXUNREG	35	X'00000023'

MQXR_* (exit reasons)

MQXR_BEFORE	1	X'00000001'
MQXR_AFTER	2	X'00000002'
MQXR_CONNECTION	3	X'00000003'

MQXE_* (environments)

MQXE_OTHER	0	X'00000000'
MQXE_MCA	1	X'00000001'
MQXE_MCA_SVRCONN	2	X'00000002'
MQXE_COMMAND_SERVER	3	X'00000003'
MQXE_MQSC	4	X'00000004'

MQ*_* (additional constants)

MQAXP_VERSION_1	1	
MQAXP_VERSION_2	2	
MQAXC_VERSION_1	1	
MQACH_VERSION_1	1	
MQAXP_CURRENT_VERSION	1	
MQAXC_CURRENT_VERSION	1	
MQACH_CURRENT_VERSION	1	
MQXACT_EXTERNAL	1	
MQXACT_INTERNAL	2	
MQXT_API_EXIT	2	
MQACH_LENGTH_1	68 (32-bit platforms) 72 (64-bit platforms) 80 (128-bit platforms)	
MQACH_CURRENT_LENGTH	68 (32-bit platforms) 72 (64-bit platforms) 80 (128-bit platforms)	

MQ*_* (null constants)

MQXPDA_NONE	X'00...00' (48 nulls)
MQXPDA_NONE_ARRAY	'\0','\0',...,'\0','\0'

MQXCC_* (completion codes)

MQXCC_FAILED	-8
--------------	----

MQRC_* (reason codes)

MQRC_API_EXIT_ERROR 2374 X'00000946'

An exit function invocation has returned an invalid response code, or has failed in some way, and the queue manager cannot determine the next action to take.

Examine both the ExitResponse and ExitResponse2 fields of the MQAXP to determine the bad response code, and change the exit to return a valid response code.

MQRC_API_EXIT_INIT_ERROR 2375 X'00000947'

The queue manager encountered an error while initializing the execution environment for an API exit function.

MQRC_API_EXIT_TERM_ERROR 2376 X'00000948'

The queue manager encountered an error while closing the execution environment for an API exit function.

MQRC_EXIT_REASON_ERROR 2377 X'00000949'

The value of the ExitReason field supplied on an exit entry point registration call (MQXEP) call is in error.

Examine the value of the ExitReason field to determine and correct the bad exit reason value.

MQRC_RESERVED_VALUE_ERROR 2378 X'0000094A'

The value of the Reserved field is in error.

Examine the value of the Reserved field to determine and correct the Reserved value.

C language typedefs

This topic provides information about typedefs associated with API exits available in C language.

Here are the C language typedefs associated with the API exits:

```
typedef PMLONG      MQPOINTER PPMQLONG;
typedef PMQBYTE     MQPOINTER PPMQBYTE;
typedef PMQHOBJS    MQPOINTER PPMQHOBJS;
typedef PMQOD       MQPOINTER PPMQOD;
typedef PMQMD       MQPOINTER PPMQMD;
typedef PMQPMO      MQPOINTER PPMQPMO;
typedef PMQGM       MQPOINTER PPMQGM;
typedef PMQCNO      MQPOINTER PPMQCNO;
typedef PMQBO       MQPOINTER PPMQBO;

typedef MQAXP       MQPOINTER PMQAXP;
typedef MQACH       MQPOINTER PMQACH;
typedef MQAXC       MQPOINTER PMQAXC;

typedef MQCHAR      MQCHAR16[16];
typedef MQCHAR16    MQPOINTER PMQCHAR16;

typedef MQLONG      MQPID;
typedef MQLONG      MQTID;
```

The exit entry point registration call (MQXEP)

Use this information to learn about MQXEP, MQXEP C language invocation, and MQXEP C function prototype.

Use the MQXEP call to:

1. Register the before and after IBM MQ API exit invocation points at which to invoke exit functions
2. Specify the exit function entry points
3. Deregister the exit function entry points

You would typically code the MQXEP calls in the MQ_INIT_EXIT exit function, but you can specify them in any subsequent exit function.

If you use an MQXEP call to register an already registered exit function, the second MQXEP call completes successfully, replacing the registered exit function.

If you use an MQXEP call to register a NULL exit function, the MQXEP call completes successfully and the exit function is deregistered.

If MQXEP calls are used to register, deregister, and reregister a particular exit function during the life of a connection request, the previously registered exit function is reactivated. Any storage still allocated and associated with this exit function instance is available for use by the exit's functions. (This storage is typically released during the invocation of the termination exit function.)

The interface to MQXEP is:

```
MQXEP (Hconfig, ExitReason, Function, EntryPoint, &ExitOpts, &CompCode, &Reason)
```

where:

Hconfig (MQHCONFIG) - input

The configuration handle, representing the API exit that includes the set of functions being initialized. This value is generated by the queue manager immediately before invoking the MQ_INIT_EXIT function, and is passed in the MQAXP to each API exit function.

ExitReason (MQLONG) - input

The reason for which the entry point is being registered, from the following reasons:

- Connection level initialization or termination (MQXR_CONNECTION)
- Before an IBM MQ API call (MQXR_BEFORE)
- After an IBM MQ API call (MQXR_AFTER)

Function (MQLONG) - input

The function identifier, valid values for which are the MQXF_* constants (see [“External constants” on page 1576](#)).

EntryPoint (PMQFUNC) - input

The address of the entry point for the exit function to be registered. The value NULL indicates either that the exit function has not been provided, or that a previous registration of the exit function is being deregistered.

ExitOpts(MQXEPO)

API exits can specify options that control how API exits are registered. If a null pointer is specified for this field, the default values of the MQXEPO structure are assumed.

CompCode (MQLONG) - output

The completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

The reason code that qualifies the completion code.

If the completion code is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If the completion code is MQCC_FAILED:

MQRC_HCONFIG_ERROR

(2280, X'8E8') The supplied configuration handle is not valid. Use the configuration handle from the MQAXP.

MQRC_EXIT_REASON_ERROR

(2377, X'949') The supplied exit function invocation reason is either not valid or is not valid for the supplied exit function identifier.

Either use one of the valid exit function invocation reasons (MQXR_* value), or use a valid function identifier and exit reason combination. (See [Table 656 on page 1579](#).)

MQRC_FUNCTION_ERROR

(2281, X'8E9') The supplied function identifier is not valid for API exit reason. The following table shows valid combinations of function identifiers and ExitReasons.

Table 656. Valid combinations of function identifiers and ExitReasons	
Function	ExitReason
MQXF_INIT MQXF_TERM	MQXR_CONNECTION

Table 656. Valid combinations of function identifiers and ExitReasons (continued)

Function	ExitReason
MQXF_CONN MQXF_CONNX MQXF_DISC MQXF_OPEN MQXF_CLOSE MQXF_PUT1 MQXF_PUT MQXF_GET MQXF_INQ MQXF_SET MQXF_BEGIN MQXF_CMIT MQXF_BACK MQXF_STAT MQXF_CB MQXF_CTL MQXF_CALLBACK MQXF_SUB MQXF_SUBRQ	MQXR_BEFORE MQXR_AFTER
MQXF_DATA_CONV_ON_GET	MQXR_BEFORE

MQRC_RESOURCE_PROBLEM

(2102, X'836') An attempt to register or deregister an exit function has failed because of a resource problem.

MQRC_UNEXPECTED_ERROR

(2195, X'893') An attempt to register or deregister an exit function has failed unexpectedly.

MQRC_PROPERTY_NAME_ERROR

(2442, X'098A') Invalid ExitProperties name.

MQRC_XEPO_ERROR

(2507, X'09CB') Exit options structure not valid.

MQXEP C language invocation

```
MQXEP (Hconfig, ExitReason, Function, EntryPoint, &ExitOpts, &CompCode, &Reason);
```

Declaration for parameter list:

```

MQHCONFIG      Hconfig;          /* Configuration handle */
MQLONG         ExitReason;       /* Exit reason */
MQLONG         Function;         /* Function identifier */
PMQFUNC        EntryPoint;       /* Function entry point */
MQXEPO         ExitOpts;        /* Options that control the action of MQXEP */
MQLONG         CompCode;        /* Completion code */
MQLONG         Reason;          /* Reason code qualifying completion
                                code */

```

MQXEP C function prototype

```

void MQXEP (
MQHCONFIG      Hconfig,          /* Configuration handle */
MQLONG         ExitReason,       /* Exit reason */
MQLONG         Function,         /* Function identifier */
PMQFUNC        EntryPoint,       /* Function entry point */
PMQXEPO        pExitOpts;       /* Options that control the action of MQXEP */
PMQLONG        pCompCode,       /* Address of completion code */

```

```
PMQLONG      pReason);          /* Address of reason code qualifying completion
                                code */
```

Exit functions

This section provides some general information to help you when using the function calls and describes how to invoke the individual exit functions.

Use this information to understand the general rules for API exit routines, and setting up and cleaning up the exit execution environment.

General rules for API exit routines

The following general rules apply when invoking API exit routines:

- In all cases, API exit functions are driven before validating API call parameters, and before any security checks (in the case of MQCONN, MQCONNX, or MQOPEN).
- The values of fields entered into and output from an exit routine are:
 - On input to a *before* IBM MQ API exit function, the value of a field can be set by the application program, or by a previous exit function invocation.
 - On output from a *before* IBM MQ API exit function, the value of a field can be left unchanged, or set to some other value by the exit function.
 - On input to an *after* IBM MQ API exit function, the value of a field can be the value set by the queue manager after processing the IBM MQ API call, or can be set to a value by a previous exit function invocation in the chain of exit functions.
 - On output from an *after* IBM MQ API call exit function, the value of a field can be left unchanged, or set to some other value by the exit function.
- Exit functions must communicate with the queue manager by using the ExitResponse and ExitResponse2 fields.
- The CompCode and Reason code fields communicate back to the application. The queue manager and exit functions can set the CompCode and Reason code fields.
- The MQXEP call returns new reason codes to the exit functions that call MQXEP. However, exit functions can translate these new reason codes to any existing reasons codes that existing and new applications can understand.
- Each exit function prototype has similar parameters to the API function with an extra level of indirection except for the CompCode and Reason.
- API exits can issue MQI calls (except MQDISC), but these MQI calls do not themselves invoke API exits.

Note, that whether the application is on a server or a client, you cannot predict the sequencing of the API exit calls. An API exit BEFORE call might not be followed immediately by an AFTER call.

The BEFORE call can be followed by another BEFORE call. For example:

```
BEFORE MQCTL
BEFORE Callback
BEFORE MQPUT
AFTER MQPUT
AFTER Callback
AFTER MQCTL
```

or

```
BEFORE XAOPEN
BEFORE MQCONNX
AFTER MQCONNX
AFTER XAOPEN
```

On the client, there is an exit that can modify the behavior of the MQCONN or MQCONNEX call, called the PreConnect exit. The PreConnect exit can modify any of the parameters on the MQCONN or MQCONNEX call including the queue manager name. The client calls this exit first and then invokes the MQCONN or MQCONNEX call. Note that only the initial MQCONN or MQCONNEX call invokes the API exit; any subsequent reconnect calls have no effect.

The execution environment

In general, all errors from exit functions are communicated back to the exit handler using the ExitResponse and ExitResponse2 fields in MQAXP.

These errors in turn are converted into MQCC_* and MQRC_* values and communicated back to the application in the CompCode and Reason fields. However, any errors encountered in the exit handler logic are communicated back to the application as MQCC_* and MQRC_* values in the CompCode and Reason fields.

If an MQ_TERM_EXIT function returns an error:

- The MQDISC call has already taken place
- There is no other opportunity to drive the *after* MQ_TERM_EXIT exit function (and thus perform exit execution environment cleanup)
- Exit execution environment cleanup is *not* performed

The exit cannot be unloaded as it might still be in use. Also, other registered exits further down in the exit chain for which the *before* exit was successful, will be driven in the reverse order.

Setting up the exit execution environment

While processing an explicit MQCONN or MQCONNEX call, exit handling logic sets up the exit execution environment before invoking the exit initialization function (MQ_INIT_EXIT). Exit execution environment setup involves loading the exit, acquiring storage for, and initializing exit parameter structures. The exit configuration handle is also allocated.

If errors occur during this phase, the MQCONN or MQCONNEX call fails with CompCode MQCC_FAILED and one of the following reason codes:

MQRC_API_EXIT_LOAD_ERROR

An attempt to load an API exit module has failed.

MQRC_API_EXIT_NOT_FOUND

An API exit function could not be found in the API exit module.

MQRC_STORAGE_NOT_AVAILABLE

An attempt to initialize the execution environment for an API exit function failed because insufficient storage was available.

MQRC_API_EXIT_INIT_ERROR

An error was encountered while initializing the execution environment for an API exit function.

Cleaning up the exit execution environment

While processing an explicit MQDISC call, or an implicit disconnect request as a result of an application ending, exit handling logic might need to clean up the exit execution environment after invoking the exit termination function (MQ_TERM_EXIT), if registered.

Cleaning up the exit execution environment involves releasing storage for exit parameter structures, possibly deleting any modules previously loaded into memory.

If errors occur during this phase, an explicit MQDISC call fails with CompCode MQCC_FAILED and the following reason code (errors are not highlighted on implicit disconnect requests):

MQRC_API_EXIT_TERM_ERROR

An error was encountered while closing the execution environment for an API exit function. The exit should *not* return any failure from the MQDISC before or after the MQ_TERM* API exit function calls.

API exits on clients

A client uses the PreConnect exit to modify the behavior of the MQCONN and MQCONNX calls and does not support API exit properties.

PreConnect exit

On a client, the PreConnect exit can be used to look up the channel definition from a central repository, such as an LDAP server.

The PreConnect exit can also modify any parameter, or all the parameters, on an MQCONN or MQCONNX call itself, for example, the queue manager name.

In the case of client applications, the PreConnect exit must be called before the API exit because the MQCONN or MQCONNX API exit is called only once the name of the queue manager is known and this name can be changed by the PreConnect exit.

Note that only the initial MQCONN or MQCONNX call invokes the exit.

API exit properties

On a server, API exits can register an MQXEPO structure at initialization time. The MQXEPO structure contains the ExitProperties field which details the group of properties the exit is interested in. This has the effect of generating a separate message property handle which the exit can manipulate separately from any application message property handle.

On a client, API exit properties are not supported. If an attempt is made to register a property group name on a client, the function fails with a reason code of MQRC_EXIT_PROPS_NOT_SUPPORTED.

Backout - MQ_BACK_EXIT

MQ_BACK_EXIT provides a backout exit function to perform *before* and *after* backout processing. Use function identifier MQXF_BACK with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* backout call exit functions.

The interface to this function is:

```
MQ_BACK_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_BACK_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_BACK_EXIT (
PMQAXP    pExitParms,    /* Address of exit parameter structure */
PMQAXC    pExitContext,  /* Address of exit context structure */
PMQHCONN  pHconn,        /* Address of connection handle */
PMQLONG   pCompCode,     /* Address of completion code */
PMQLONG   pReason);      /* Address of reason code qualifying completion
                           code */
```

Begin - MQ_BEGIN_EXIT

MQ_BEGIN_EXIT provides a begin exit function to perform *before* and *after* MQBEGIN call processing. Use function identifier MQXF_BEGIN with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQBEGIN call exit functions.

The interface to this function is:

```
MQ_BEGIN_EXIT (&ExitParms, &ExitContext, &Hconn, &pBeginOptions, &CompCode,
               &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pBeginOptions (PMQBO)- input/output

Pointer to begin options.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQBO    pBeginOptions;  /* Ptr to begin options */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_BEGIN_EXIT (&ExitParms, &ExitContext, &Hconn, &pBeginOptions, &CompCode,
               &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_BEGIN_EXIT (
PMQAXP    pExitParms,      /* Address of exit parameter structure */
PMQAXC    pExitContext,    /* Address of exit context structure */
PMQHCONN  pHconn,          /* Address of connection handle */
PPMQBO    ppBeginOptions,  /* Address of ptr to begin options */
PMQLONG   pCompCode,       /* Address of completion code */
PMQLONG   pReason);        /* Address of reason code qualifying completion
                             code */
```

Callback - MQ_CALLBACK_EXIT

MQ_CALLBACK_EXIT provides an exit function to perform *before* and *after* callback processing. Use function identifier MQXF_CALLBACK with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* callback call exit functions.

The interface to this function is:

```
MQ_CALLBACK_EXIT (&ExitParms, &ExitContext, &Hconn, &pMsgDesc, &pGetMsgOpts,
                  &pBuffer, &pMQCBContext)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure

ExitContext (MQAXC) - input/output

Exit context structure

Hconn (MQHCONN) - input/output

Connection handle

pMsgDesc

Message descriptor

pGetMsgOpts

Options that control the action of MQGET

pBuffer

Area to contain the message data

pMQCBContext

Context data for the callback

C language invocation

The queue manager logically defines the following variables:

```

MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQMD    pMsgDesc;       /* Message descriptor */
PMQGM0   pGetMsgOpts;    /* Options that define the operation of the consumer */
PMQVOID  pBuffer;        /* Area to contain the message data */
PMQCBC   pContext;       /* Context data for the callback */

```

The queue manager then logically calls the exit as follows:

```

MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pMsgDesc, &pGetMsgOpts, &pBuffer,
               &pContext);

```

Your exit must match the following C function prototype:

```

void MQENTRY MQ_CALLBACK_EXIT (
PMQAXP    pExitParms;    /* Exit parameter structure */
PMQAXC    pExitContext;  /* Exit context structure */
PMQHCONN  pHconn;        /* Connection handle */
PPMQMD    ppMsgDesc;     /* Message descriptor */
PPMQGM0   ppGetMsgOpts;  /* Options that define the operation of the consumer */
PPMVOID   ppBuffer;      /* Area to contain the message data */
PPMCBC    ppContext;     /* Context data for the callback */
)

```

Usage notes

1. The Callback exit is invoked before the consumer is invoked and after the consumer's consumer function has completed. Although the MQMD and MQGMO structures are alterable, changing the values in the before exit does not redrive the retrieval of a message from the queue as the message has already been removed from the queue to be delivered to the consumer function

Manage callback functions - MQ_CB_EXIT

MQ_CB_EXIT provides an exit function to perform *before* and *after* the MQCB call. Use function identifier MQXF_CB with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQCB call exit functions.

The interface to this function is:

```

MQ_CB_EXIT (&ExitParms, &ExitContext, &Hconn, &Operation, &pCallbackDesc,
            &Hobj, &pMsgDesc, &pGetMsgOpts, &CompCode, &Reason)

```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure

ExitContext (MQAXC) - input/output

Exit context structure

Hconn (MQHCONN) - input/output

Connection handle

Operation (MQLONG) - input/output

Operation value

pCallbackDesc (PMQCBD) - input/output

Callback descriptor

Hobj (MQHOBJ) - input/output

Object handle

pMsgDesc (PMQMD) - input/output

Message descriptor

pGetMsgOpts (PMQGM0) - input/output

Options that control the action of MQCB

CompCode (MQLONG) - input/output

Completion code

Reason (MQLONG) - input/output

Reason code qualifying CompCode

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN   Hconn;         /* Connection handle */
MQLONG    Operation;     /* Operation value. */
MQCBD     pMsgDesc;      /* Callback descriptor. */
MQHOBJS   Hobj;         /* Object handle. */
PMQMD     pMsgDesc;      /* Message descriptor */
PMQGM0    pGetMsgOpts;   /* Options that define the operation of the consumer */
PMQLONG   CompCode;      /* Completion code. */
PMQLONG   Reason;        /* Reason code qualifying CompCode. */
```

The queue manager then logically calls the exit as follows:

```
MQ_CB_EXIT (&ExitParms, &ExitContext, &Hconn, &Operation, &Hobj, &pMsgDesc,
            &pGetMsgOpts, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CB_EXIT (
PMQAXP    pExitParms;    /* Exit parameter structure */
PMQAXC    pExitContext;  /* Exit context structure */
PMQHCONN   pHconn;       /* Connection handle */
MQLONG     pOperation;   /* Callback operation */
PMQHOBJS   pHobj;        /* Object handle */
PPMQMD     ppMsgDesc;    /* Message descriptor */
PPMQGM0    ppGetMsgOpts; /* Options that control the action of MQCB */
PMQLONG     pCompCode;   /* Completion code */
PMQLONG     pReason;     /* Reason code qualifying CompCode */
```

Close - MQ_CLOSE_EXIT

MQ_CLOSE_EXIT provides a close exit function to perform *before* and *after* MQCLOSE call processing. Use function identifier MQXF_CLOSE with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQCLOSE call exit functions.

The interface to this function is:

```
MQ_CLOSE_EXIT (&ExitParms, &ExitContext, &Hconn, &pHobj,
               &Options, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pHobj (PMQHOBJS) - input

Pointer to object handle.

Options (MQLONG)- input/output

Close options.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;      /* Exit parameter structure */
MQAXC      ExitContext;    /* Exit context structure */
MQHCONN    Hconn;          /* Connection handle */
PMQHOBj    pHobj;          /* Ptr to object handle */
MQLONG     Options;        /* Close options */
MQLONG     CompCode;       /* Completion code */
MQLONG     Reason;        /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CLOSE_EXIT (&ExitParms, &ExitContext, &Hconn, &pHobj, &Options,
               &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CLOSE_EXIT (
PMQAXP      pExitParms,    /* Address of exit parameter structure */
PMQAXC      pExitContext,  /* Address of exit context structure */
PMQHCONN    pHconn,        /* Address of connection handle */
PPMHOBj     ppHobj,        /* Address of ptr to object handle */
PMQLONG     pOptions,      /* Address of close options */
PMQLONG     pCompCode,     /* Address of completion code */
PMQLONG     pReason);      /* Address of reason code qualifying
                             completion code */
```

Commit - MQ_CMtT_EXIT

MQ_CMtT_EXIT provides a commit exit function to perform *before* and *after* commit processing. Use function identifier MQXF_CMtT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* commit call exit functions.

If a commit operation fails, and the transaction is backed out, the MQCMIT call fails with MQCC_WARNING and MQRC_BACKED_OUT. These return and reason codes are passed into any *after* MQCMIT exit functions to give the exit functions an indication that the unit of work has been backed out.

The interface to this function is:

```
MQ_CMtT_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CMIT_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CMIT_EXIT (
PMQAXP    pExitParms,      /* Address of exit parameter structure */
PMQAXC    pExitContext,    /* Address of exit context structure */
PMQHCONN  pHconn,         /* Address of connection handle */
PMQLONG   pCompCode,       /* Address of completion code */
PMQLONG   pReason);        /* Address of reason code qualifying completion
                             code */
```

Usage notes

1. The MQ_GET_EXIT function interface described here is used for both the MQXF_GET exit function and the [“MQXF_DATA_CONV_ON_GET”](#) on page 1595 exit function.

Separate entry points are defined for these two exit functions, so to intercept *both* the MQXEP call must be used twice; for this call use function identifier MQXF_GET.

Because the MQ_GET_EXIT interface is the same for MQXF_GET and MQXF_DATA_CONV_ON_GET, a single exit function can be used for both; the *Function* field in the MQAXP structure indicates which exit function has been invoked. Alternatively, the MQXEP call can be used to register different exit functions for the two cases.

Connect and connect extension - MQ_CONNX_EXIT

MQ_CONNX_EXIT provides:

- Connection exit function to perform *before* and *after* MQCONN processing
- Connection extension exit function to perform *before* and *after* MQCONNX processing

The same interface, described here, is invoked for both MQCONN and MQCONNX call exit functions.

When the message channel agent (MCA) responds to an inbound client connection, the MCA can connect and make a number of IBM MQ API calls before the client state is fully known. These API calls call the

API exit functions with the MQAXC based on the MCA program itself (for example in the UserId and ConnectionName fields of the MQAXC).

When the MCA responds to subsequent inbound client API calls, the MQAXC structure is based on the inbound client, setting the UserId and ConnectionName fields appropriately.

The queue manager name set by the application on an MQCONN or MQCONNEX call is passed to the underlying connect call. Any attempt by a *before* MQ_CONNEX_EXIT to change the name of the queue manager has no effect.

Use function identifiers MQXF_CONN and MQXF_CONNEX with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQCONN and MQCONNEX call exit functions.

An MQ_CONNEX_EXIT exit called for reason MQXR_BEFORE *must not* issue any IBM MQ API calls, as the correct environment has not been set up at this time.

An MQ_CONNEX_EXIT cannot call MQDISC from an API exit call for the connection for which it is being called. This restriction is applicable to both client and server API exits.

The interface to MQCONN and MQCONNEX is identical:

```
MQ_CONNEX_EXIT (&ExitParms, &ExitContext, &pQMGrName, &pConnectOpts,  
&pHconn, &CompCode, &Reason);
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

pQMGrName (PMQCHAR) - input

Pointer to the queue manager name supplied on the MQCONNEX call. The exit must not change this name on the MQCONN or MQCONNEX call.

pConnectOpts (PMQCNO) - input/output

Pointer to the options that control the action of the MQCONNEX call.

See [“MQCNO - Connect options” on page 306](#) for details.

For exit function MQXF_CONN, pConnectOpts points to the default connect options structure (MQCNO_DEFAULT).

pHconn (PMQHCONN) - input

Pointer to the connection handle.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion)

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;      /* Exit parameter structure */
MQAXC      ExitContext;    /* Exit context structure */
PMQCHAR    pQMgrName;     /* Ptr to Queue manager name */
PMQCNO     pConnectOpts;  /* Ptr to Connection options */
PMQHCONN   pHconn;        /* Ptr to Connection handle */
MQLONG     CompCode;       /* Completion code */
MQLONG     Reason;        /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CONNX_EXIT (&ExitParms, &ExitContext, &pQMgrName, &pConnectOpts,
               &pHconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CONNX_EXIT (
PMQAXP      pExitParms,    /* Address of exit parameter structure */
PMQAXC      pExitContext,  /* Address of exit context structure */
PPMQCHAR    ppQMgrName,   /* Address of ptr to queue manager name */
PPMCNO      ppConnectOpts, /* Address of ptr to connection options */
PPMHCONN    ppHconn,      /* Address of ptr to connection handle */
PMQLONG     pCompCode,    /* Address of completion code */
PMQLONG     pReason);     /* Address of reason code qualifying
                           completion code */
```

Usage notes

1. The MQ_CONNX_EXIT function interface described here is used for both the MQCONN call and the MQCONNX call. However, separate entry points are defined for these two calls. To intercept *both* calls, the MQXEP call must be used at least twice - once with function identifier MQXF_CONN, and again with MQXF_CONNX.

Because the MQ_CONNX_EXIT interface is the same for MQCONN and MQCONNX, a single exit function can be used for both calls; the *Function* field in the MQAXP structure indicates which call is in progress. Alternatively, the MQXEP call can be used to register different exit functions for the two calls.

2. When a message channel agent (MCA) responds to an inbound client connection, the MCA can issue a number of MQ calls before the client state is fully known. These MQ calls result in the API exit functions being invoked with the MQAXC structure containing data relating to the MCA, and not to the client (for example, user identifier and connection name). However, once the client state is fully known, subsequent MQ calls result in the API exit functions being invoked with the appropriate client data in the MQAXC structure.
3. All MQXR_BEFORE exit functions are invoked before any parameter validation is performed by the queue manager. The parameters might therefore be invalid (including invalid pointers for the addresses of parameters).

The MQ_CONNX_EXIT function is invoked before any authorization checks are performed by the queue manager.

4. The exit function must not change the name of the queue manager specified on the MQCONN or MQCONNX call. If the name is changed by the exit function, the results are undefined.
5. An MQXR_BEFORE exit function for the MQ_CONNX_EXIT cannot issue MQ calls other than MQXEP.

Control callback - MQ_CTL_EXIT

MQ_CTL_EXIT provides a subscription request exit function to perform *before* and *after* control callback processing. Use function identifier MQXF_CTL with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* control callback call exit functions.

The interface to this function is:

```
MQ_CTL_EXIT (&Hconn, &Operation, &ControlOpts, &CompCode, &Reason)
```

where the parameters are:

Hconn (MQHCONN) - input/output

Connection handle.

Operation (MQLONG) input/output

The operation being processed on the callback defined for the specified object handle

ControlOpts (MQCTLO) input/output

Options that control the action of MQCTL

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQHCONN  Hconn;          /* Connection handle */
MQLONG   Operation;      /* Operation being processed */
MQCTLO   ControlOpts;    /* Options that control the action of MQCTL */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CTL_EXIT (&Hconn, &Operation, &ControlOpts, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CTL_EXIT (
PMQHCONN pHconn;          /* Address of connection handle */
PMQLONG  pOperation;      /* Address of operation being processed */
PMQCTLO  pControlOpts;    /* Address of options that control the action of MQCTL */
PMQLONG  pCompCode;       /* Address of completion code */
PMQLONG  pReason;         /* Address of reason code qualifying completion code */
```

Disconnect - MQ_DISC_EXIT

MQ_DISC_EXIT provides a disconnect exit function to perform *before* and *after* MQDISC exit processing. Use function identifier MQXF_DISC with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQDISC call exit functions.

The interface to this function is

```
MQ_DISC_EXIT (&ExitParms, &ExitContext, &pHconn,
&CompCode, &Reason);
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

pHconn (PMQHCONN) - input

Pointer to the connection handle.

For the before MQDISC call, the value of this field is one of:

- The connection handle returned on the MQCONN or MQCONNEX call
- Zero, for environments where an environment-specific adapter has connected to the queue manager
- A value set by a previous exit function invocation

For the after MQDISC call, the value of this field is zero or a value set by a previous exit function invocation.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

MQAXP	ExitParms;	/* Exit parameter structure */
MQAXC	ExitContext;	/* Exit context structure */
PMQHCONN	pHconn;	/* Ptr to Connection handle */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code */

The queue manager then logically calls the exit as follows:

```
MQ_DISC_EXIT (&ExitParms, &ExitContext, &pHconn,
&CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_DISC_EXIT (
PMQAXP      pExitParms,      /* Address of exit parameter structure */
PMQAXC      pExitContext,    /* Address of exit context structure */
PPMQHCONN   ppHconn,         /* Address of ptr to connection handle */
PMQLONG     pCompCode,       /* Address of completion code */
```

```
PMQLONG      pReason);      /* Address of reason code qualifying
                             completion code */
```

Get - MQ_GET_EXIT

MQ_GET_EXIT provides a get exit function to perform *before* and *after* MQGET call processing.

There are two function identifiers:

1. Use MQXF_GET with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQGET call exit functions.
2. See “MQXF_DATA_CONV_ON_GET” on page 1595 for information on using the MQXF_DATA_CONV_ON_GET function identifier.

The interface to this function is:

```
MQ_GET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pGetMsgOpts, &BufferLength, &pBuffer, &pDataLength,
             &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

Hobj (MQHOBJ) - input/output

Object handle.

pMsgDesc (PMQMD) - input/output

Pointer to message descriptor.

pGetMsgOpts (PMQGMO) - input/output

Pointer to get message options.

BufferLength (MQLONG) - input/output

Message buffer length.

pBuffer (PMQBYTE) - input/output

Pointer to message buffer.

pDataLength (PMQLONG) - input/output

Pointer to data length field.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;      /* Exit parameter structure */
MQAXC      ExitContext;    /* Exit context structure */
MQHCONN    Hconn;         /* Connection handle */
MQHOBJ     Hobj;          /* Object handle */
PMQMD      pMsgDesc;       /* Ptr to message descriptor */
PMQPMO     pGetMsgOpts;    /* Ptr to get message options */
MQLONG     BufferLength;    /* Message buffer length */
PMQBYTE    pBuffer;        /* Ptr to message buffer */
PMQLONG    pDataLength;    /* Ptr to data length field */
MQLONG     CompCode;       /* Completion code */
MQLONG     Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_GET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pGetMsgOpts, &BufferLength, &pBuffer, &pDataLength,
             &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_GET_EXIT (
    PMQAXP      pExitParms,      /* Address of exit parameter structure */
    PMQAXC      pExitContext,    /* Address of exit context structure */
    PMQHCONN    pHconn,         /* Address of connection handle */
    PMQHOBJ     pHobj,          /* Address of object handle */
    PPMQMD      ppMsgDesc,       /* Address of ptr to message descriptor */
    PPMQGMO     ppGetMsgOpts,    /* Address of ptr to get message options */
    PMQLONG     pBufferLength,   /* Address of message buffer length */
    PPMQBYTE    ppBuffer,        /* Address of ptr to message buffer */
    PPMQLONG    ppDataLength,    /* Address of ptr to data length field */
    PMQLONG     pCompCode,       /* Address of completion code */
    PMQLONG     pReason);        /* Address of reason code qualifying
                                completion code */
```

Usage notes

1. The MQ_GET_EXIT function interface described here is used for both the MQXF_GET exit function and the “MQXF_DATA_CONV_ON_GET” on page 1595 exit function.

Separate entry points are defined for these two exit functions, so to intercept *both* the MQXEP call must be used twice; for this call use function identifier MQXF_GET.

Because the MQ_GET_EXIT interface is the same for MQXF_GET and MQXF_DATA_CONV_ON_GET, a single exit function can be used for both; the *Function* field in the MQAXP structure indicates which exit function has been invoked. Alternatively, the MQXEP call can be used to register different exit functions for the two cases.

MQXF_DATA_CONV_ON_GET

See [MQ_GET_EXIT](#) for information about the interface to this call, and a sample C language declaration.

Usage notes

If registered, this entry point is called when messages arrive at the application but before any data conversion has occurred. This can be useful if the API exit needs to perform processing, such as decryption or decompression, before the message is passed to data conversion. The exit can, if necessary, cause data conversion to be bypassed by returning MQXCC_SUPPRESS_FUNCTION; for more information, see [MQAXP](#) structure.

Registering for this entry point on a client has the effect of causing the data conversion to be performed locally on the client machine. For correct operation it might, therefore, be necessary to install the application conversion exits on the client. Note that MQXF_DATA_CONV_ON_GET is also used for asynchronous consume.

When using the `MQ_GET_EXIT` call, use `MQXF_DATA_CONV_ON_GET`, with exit reason `MQXR_BEFORE`, to register a *before* MQGET data conversion exit function.

There is no `MQXR_AFTER` exit function for `MQXF_DATA_CONV_ON_GET`; the `MQXR_AFTER` exit function for `MQXF_GET` provides the required capability for exit processing after data conversion.

Separate entry points are defined for the `MQ_GET_EXIT` call, so to intercept *both* exit functions, the `MQXEP` call must be used twice; for this call use function identifier `MQXF_DATA_CONV_ON_GET`.

Because the `MQ_GET_EXIT` interface is the same for `MQXF_GET` and `MQXF_DATA_CONV_ON_GET`, a single exit function can be used for both; the *Function* field in the `MQAXP` structure indicates which exit function has been invoked. Alternatively, the `MQXEP` call can be used to register different exit functions for the two cases.

Initialization - MQ_INIT_EXIT

`MQ_INIT_EXIT` provides connection level initialization, indicated by setting `ExitReason` in `MQAXP` to `MQXR_CONNECTION`.

During the initialization, note the following:

- The `MQ_INIT_EXIT` function calls `MQXEP` to register the IBM MQ API verbs and the `ENTRY` and `EXIT` points in which it is interested.
- Exits do not need to intercept all the IBM MQ API verbs. Exit functions are invoked only if an interest has been registered.
- Storage that is to be used by the exit can be acquired while initializing it.
- If a call to this function fails, the `MQCONN` or `MQCONNEX` call that invoked it also fails with a `CompCode` and `Reason` that depend on the value of the `ExitResponse` field in `MQAXP`.
- An `MQ_INIT_EXIT` exit must not issue IBM MQ API calls, because the correct environment has not been set up at this time.
- If an `MQ_INIT_EXIT` fails with `MQXCC_FAILED`, the queue manager returns from the `MQCONN` or `MQCONNEX` call that called it with `MQCC_FAILED` and `MQRC_API_EXIT_ERROR`.
- If the queue manager encounters an error while initializing the API exit function execution environment before invoking the first `MQ_INIT_EXIT`, the queue manager returns from the `MQCONN` or `MQCONNEX` call that invoked `MQ_INIT_EXIT` with `MQCC_FAILED` and `MQRC_API_EXIT_INIT_ERROR`.

The interface to `MQ_INIT_EXIT` is:

```
MQ_INIT_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

CompCode (MQLONG) - input/output

Pointer to completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Pointer to reason code qualifying the completion code.

If the completion code is `MQCC_OK`, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

The CompCode and Reason returned to the application depend on the value of the ExitResponse field in MQAXP.

C language invocation

The queue manager logically defines the following variables:

MQAXP	ExitParms;	/* Exit parameter structure */
MQAXC	ExitContext;	/* Exit context structure */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code */

The queue manager then logically calls the exit as follows:

```
MQ_INIT_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_INIT_EXIT (
    PMQAXP      pExitParms,      /* Address of exit parameter structure */
    PMQAXC      pExitContext,    /* Address of exit context structure */
    PMQLONG     pCompCode,       /* Address of completion code */
    PMQLONG     pReason);        /* Address of reason code qualifying
                                completion code */
```

Usage notes

1. The MQ_INIT_EXIT function can issue the MQXEP call to register the addresses of the exit functions for the particular MQ calls to be intercepted. It is not necessary to intercept all MQ calls, or to intercept both MQXR_BEFORE and MQXR_AFTER calls. For example, an exit suite could choose to intercept only the MQXR_BEFORE call of MQPUT.
2. Storage that is to be used by exit functions in the exit suite can be acquired by the MQ_INIT_EXIT function. Alternatively, exit functions can acquire storage when they are invoked, as and when needed. However, all storage should be freed before the exit suite is terminated; the MQ_TERM_EXIT function can free the storage, or an exit function invoked earlier.
3. If MQ_INIT_EXIT returns MQXCC_FAILED in the *ExitResponse* field of MQAXP, or fails in some other way, the MQCONN or MQCONNX call that caused MQ_INIT_EXIT to be invoked also fails, with the *CompCode* and *Reason* parameters set to appropriate values.
4. An MQ_INIT_EXIT function cannot issue MQ calls other than MQXEP.

Inquire - MQ_INQ_EXIT

MQ_INQ_EXIT provides an inquire exit function to perform *before* and *after* MQINQ call processing. Use function identifier MQXF_INQ with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQINQ call exit functions.

The interface to this function is:

```
MQ_INQ_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

Hobj (MQHOBJ) - input

Object handle.

SelectorCount (MQLONG) - input

Count of selectors

pSelectors (PMQLONG) - input/output

Pointer to array of selector values.

IntAttrCount (MQLONG) - input

Count of integer attributes.

pIntAttrs (PMQLONG) - input/output

Pointer to array of integer attribute values.

CharAttrLength (MQLONG) - input/output

Character attributes array length.

pCharAttrs (PMQCHAR) - input/output

Pointer to character attributes array.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;          /* Exit parameter structure */
MQAXC      ExitContext;        /* Exit context structure */
MQHCONN    Hconn;              /* Connection handle */
MQHOBJ     Hobj;               /* Object handle */
MQLONG     SelectorCount;      /* Count of selectors */
PMQLONG    pSelectors;         /* Ptr to array of attribute selectors */
MQLONG     IntAttrCount;       /* Count of integer attributes */
PMQLONG    pIntAttrs;          /* Ptr to array of integer attributes */
MQLONG     CharAttrLength;     /* Length of char attributes array */
PMQCHAR    pCharAttrs;         /* Ptr to character attributes */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_INQ_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_INQ_EXIT (
    PMQAXP    pExitParms,      /* Address of exit parameter structure */
    PMQAXC    pExitContext,    /* Address of exit context structure */
    PMQHCONN  pHconn,          /* Address of connection handle */
    PMQHOBJS  pHobj,           /* Address of object handle */
    PMQLONG   pSelectorCount,   /* Address of selector count */
    PPMQLONG  ppSelectors,      /* Address of ptr to array of selectors */
    PMQLONG   pIntAttrCount;    /* Address of count of integer attributes */
    PPMQLONG  ppIntAttrs,       /* Address of ptr to array of integer attributes */
    PMQLONG   pCharAttrLength,  /* Address of character attribute length */
    PPMQCHAR  ppCharAttrs,      /* Address of ptr to character attributes array */
    PMQLONG   pCompCode,        /* Address of completion code */
    PMQLONG   pReason;          /* Address of reason code qualifying completion
                                code */
)
```

Open - MQ_OPEN_EXIT

MQ_OPEN_EXIT provides an open exit function to perform *before* and *after* MQOPEN call processing. Use function identifier MQXF_OPEN with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQOPEN call exit functions.

The interface to this function is

```
MQ_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &Options,
              &pHobj, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pObjDesc (PMQOD) - input/output

Pointer to object descriptor.

Options (MQLONG) - input/output

Open options.

pHobj (PMQHOBJS) - input

Pointer to object handle.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;      /* Exit parameter structure */
MQAXC      ExitContext;    /* Exit context structure */
MQHCONN    Hconn;         /* Connection handle */
PMQOD      pObjDesc;      /* Ptr to object descriptor */
MQLONG     Options;       /* Open options */
MQHOBJ     pHobj;         /* Ptr to object handle */
MQLONG     CompCode;      /* Completion code */
MQLONG     Reason;        /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &Options,
              &pHobj, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_OPEN_EXIT (
PMQAXP      pExitParms,    /* Address of exit parameter structure */
PMQAXC      pExitContext,  /* Address of exit context structure */
PMQHCONN    pHconn,       /* Address of connection handle */
PPMQOD      ppObjDesc,    /* Address of ptr to object descriptor */
PMQLONG     pOptions,      /* Address of open options */
PPMQHOBJS   ppHobj,       /* Address of ptr to object handle */
PMQLONG     pCompCode,     /* Address of completion code */
PMQLONG     pReason);     /* Address of reason code qualifying
                           completion code */
```

Put - MQ_PUT_EXIT

MQ_PUT_EXIT provides a put exit function to perform *before* and *after* MQPUT call processing. Use function identifier MQXF_PUT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQPUT call exit functions.

The interface to this function is:

```
MQ_PUT_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

Hobj (MQHOBJS) - input/output

Object handle.

pMsgDesc (PMQMD) - input/output

Pointer to message descriptor.

pPutMsgOpts (PMQPMO) - input/output

Pointer to put message options.

BufferLength (MQLONG) - input/output

Message buffer length.

pBuffer (PMQBYTE) - input/output

Pointer to message buffer.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

MQAXP	ExitParms;	/* Exit parameter structure */
MQAXC	ExitContext;	/* Exit context structure */
MQHCONN	Hconn;	/* Connection handle */
MQHOBJ	Hobj;	/* Object handle */
PMQMD	pMsgDesc;	/* Ptr to message descriptor */
PMQPMO	pPutMsgOpts;	/* Ptr to put message options */
MQLONG	BufferLength;	/* Message buffer length */
PMQBYTE	pBuffer;	/* Ptr to message data */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code */

The queue manager then logically calls the exit as follows:

```
MQ_PUT_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_PUT_EXIT (
PMQAXP      pExitParms,      /* Address of exit parameter structure */
PMQAXC      pExitContext,    /* Address of exit context structure */
PMQHCONN    pHconn,         /* Address of connection handle */
PMQHOBJ     pHobj,          /* Address of object handle */
PPMQMD      ppMsgDesc,       /* Address of ptr to message descriptor */
PPMQPMO     ppPutMsgOpts,    /* Address of ptr to put message options */
PMQLONG     pBufferLength,   /* Address of message buffer length */
PPMQBYTE    ppBuffer,        /* Address of ptr to message buffer */
PMQLONG     pCompCode,       /* Address of completion code */
PMQLONG     pReason);        /* Address of reason code qualifying
                               completion code */
```

Usage notes

- Report messages generated by the queue manager skip the normal call processing. As a result, such messages cannot be intercepted by the MQ_PUT_EXIT function or the MQPUT1 function. However, report messages generated by the message channel agent are processed normally, and hence can be intercepted by the MQ_PUT_EXIT function or the MQ_PUT1_EXIT function. To be sure to intercepting all of the report messages generated by the MCA, both MQ_PUT_EXIT and MQ_PUT1_EXIT should be used.

Put1 - MQ_PUT1_EXIT

MQ_PUT1_EXIT provides a *put one message only* exit function to perform *before* and *after* MQPUT1 call processing. Use function identifier MQXF_PUT1 with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQPUT1 call exit functions.

The interface to this function is:

```
MQ_PUT1_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &pMsgDesc,  
&pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pObjDesc (PMQOD) - input/output

Pointer to object descriptor.

pMsgDesc (PMQMD) - input/output

Pointer to message descriptor.

pPutMsgOpts (PMQPMO) - input/output

Pointer to put message options.

BufferLength (MQLONG) - input/output

Message buffer length.

pBuffer (PMQBYTE) - input/output

Pointer to message buffer.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

MQAXP	ExitParms;	/* Exit parameter structure */
MQAXC	ExitContext;	/* Exit context structure */
MQHCONN	Hconn;	/* Connection handle */
PMQOD	pObjDesc;	/* Ptr to object descriptor */
PMQMD	pMsgDesc;	/* Ptr to message descriptor */
PMQPMO	pPutMsgOpts;	/* Ptr to put message options */
MQLONG	BufferLength;	/* Message buffer length */
PMQBYTE	pBuffer;	/* Ptr to message data */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code */

The queue manager then logically calls the exit as follows:

```
MQ_PUT1_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &pMsgDesc,
              &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_PUT1_EXIT (
    PMQAXP      pExitParms,      /* Address of exit parameter structure */
    PMQAXC      pExitContext,    /* Address of exit context structure */
    PMQHCONN    pHconn,         /* Address of connection handle */
    PPMQOD      ppObjDesc,       /* Address of ptr to object descriptor */
    PPMQMD      ppMsgDesc,       /* Address of ptr to message descriptor */
    PPMQPMO     ppPutMsgOpts,    /* Address of ptr to put message options */
    PMQLONG     pBufferLength,   /* Address of message buffer length */
    PMQBYTE     ppBuffer,        /* Address of ptr to message buffer */
    PMQLONG     pCompCode,       /* Address of completion code */
    PMQLONG     pReason);        /* Address of reason code qualifying
                                completion code */
```

Set - MQ_SET_EXIT

MQ_SET_EXIT provides a set exit function to perform *before* and *after* MQSET call processing. Use function identifier MQXF_SET with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQSET call exit functions.

The interface to this function is:

```
MQ_SET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttr, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

Hobj (MQHOBJ) - input

Object handle.

SelectorCount (MQLONG) - input

Count of selectors

pSelectors (PMQLONG) - input/output

Pointer to array of selector values.

IntAttrCount (MQLONG) - input

Count of integer attributes.

pIntAttrs (PMQLONG) - input/output

Pointer to array of integer attribute values.

CharAttrLength (MQLONG) - input/output

Character attributes array length.

pCharAttrs (PMQCHAR) - input/output

Pointer to character attribute values.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;          /* Exit parameter structure */
MQAXC      ExitContext;        /* Exit context structure */
MQHCONN    Hconn;             /* Connection handle */
MQHOBJ     Hobj;              /* Object handle */
MQLONG     SelectorCount;      /* Count of selectors */
PMQLONG    pSelectors;         /* Ptr to array of attribute selectors */
MQLONG     IntAttrCount;       /* Count of integer attributes */
PMQLONG    pIntAttrs;          /* Ptr to array of integer attributes */
MQLONG     CharAttrLength;     /* Length of char attributes array */
PMQCHAR    pCharAttrs;         /* Ptr to character attributes */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;            /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_SET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_SET_EXIT (
PMQAXP      pExitParms,        /* Address of exit parameter structure */
PMQAXC      pExitContext,      /* Address of exit context structure */
PMQHCONN    pHconn,           /* Address of connection handle */
PMQHOBJ     pHobj,            /* Address of object handle */
PMQLONG     pSelectorCount,    /* Address of selector count */
PPMQLONG    ppSelectors,       /* Address of ptr to array of selectors */
PMQLONG     pIntAttrCount;     /* Address of count of integer attributes */
PPMQLONG    ppIntAttrs,        /* Address of ptr to array of integer attributes */
PMQLONG     pCharAttrLength,   /* Address of character attribute length */
PPMQCHAR    ppCharAttrs,       /* Address of ptr to character attributes array */
PMQLONG     pCompCode,         /* Address of completion code */
PMQLONG     pReason);          /* Address of reason code qualifying completion
                                code */
```

Status - MQ_STAT_EXIT

MQ_STAT_EXIT provides a status exit function to perform *before* and *after* MQSTAT call processing. Use function identifier MQXF_STAT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQSTAT call exit functions.

The interface to this function is:

```
MQ_STAT_EXIT (&ExitParms, &ExitContext, &Hconn, &Type, &pStatus
              &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

Type (MQLONG) - input

Type of status information to retrieve.

pStatus (PMQSTS) - output

Pointer to status buffer.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

Your exit must match the following C function prototype:

```
void MQENTRY MQ_STAT_EXIT (
    PMQAXP    pExitParms,      /* Address of exit parameter structure */
    PMQAXC    pExitContext,    /* Address of exit context structure */
    PMQHCONN  pHconn,         /* Address of connection handle */
    PMQLONG   pType,           /* Address of status type */
    PPMQSTS   ppStatus,        /* Address of status buffer */
    PMQLONG   pCompCode,       /* Address of completion code */
    PMQLONG   pReason);        /* Address of reason code qualifying completion
                                code */
```

Termination - MQ_TERM_EXIT

MQ_TERM_EXIT provides connection level termination, registered with a function identifier of MQXF_TERM and ExitReason MQXR_CONNECTION. If registered, MQ_TERM_EXIT is called once for every disconnect request.

As part of the termination, storage no longer required by the exit can be released, and any clean up required can be performed.

If an MQ_TERM_EXIT fails with MQXCC_FAILED, the queue manager returns from the MQDISC that called it with MQCC_FAILED and MQRC_API_EXIT_ERROR.

If the queue manager encounters an error while terminating the API exit function execution environment after invoking the last MQ_TERM_EXIT, the queue manager returns from the MQDISC call that invoked MQ_TERM_EXIT with MQCC_FAILED and MQRC_API_EXIT_TERM_ERROR

The interface to this function is:

```
MQ_TERM_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED, the exit function can set the reason code field to any valid MQRC_* value.

The CompCode and Reason returned to the application depend on the value of the ExitResponse field in MQAXP.

C language invocation

The queue manager logically defines the following variables:

MQAXP	ExitParms;	/* Exit parameter structure */
MQAXC	ExitContext;	/* Exit context structure */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code */

The queue manager then logically calls the exit as follows:

```
MQ_TERM_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_TERM_EXIT (
PMQAXP      pExitParms,      /* Address of exit parameter structure */
PMQAXC      pExitContext,    /* Address of exit context structure */
PMQLONG     pCompCode,       /* Address of completion code */
PMQLONG     pReason);        /* Address of reason code qualifying
                               completion code */
```

Usage notes

1. The MQ_TERM_EXIT function is optional. It is not necessary for an exit suite to register a termination exit if there is no termination processing to be done.

If functions belonging to the exit suite acquire resources during the connection, an MQ_TERM_EXIT function is a convenient point at which to free those resources, for example, freeing storage obtained dynamically.
2. If an MQ_TERM_EXIT function is registered when the MQDISC call is issued, the exit function is invoked after all of the MQDISC exit functions have been invoked.
3. If MQ_TERM_EXIT returns MQXCC_FAILED in the *ExitResponse* field of MQAXP, or fails in some other way, the MQDISC call that caused MQ_TERM_EXIT to be invoked also fails, with the *CompCode* and *Reason* parameters set to appropriate values.

Register subscription - MQ_SUB_EXIT

MQ_SUB_EXIT provides an exit function to perform *before* and *after* subscription reregistration processing. Use function identifier MQXF_SUB with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* subscription registration call exit functions.

The interface to this function is:

```
MQ_SUB_EXIT (&ExitParms, &ExitContext, &Hconn, &pSubDesc, &pHobj, &pHsub, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input/output

Connection handle.

pSubDesc - input/output

Array of attribute selectors.

pHobj - input/output

Object handle

pHsub (MQHOBJ) input/output

Subscription handle

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP      ExitParms;      /* Exit parameter structure */
MQAXC      ExitContext;    /* Exit context structure */
MQHCONN    Hconn;          /* Connection handle */
PMQSD      pSubDesc;       /* Subscription descriptor */
PMQHOBj    pHobj;          /* Object Handle */
PMQHOBj    pHsub;          /* Subscription handle */
MQLONG     CompCode;       /* Completion code */
MQLONG     Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_SUB_EXIT (&ExitParms, &ExitContext, &Hconn, &pSubDesc, &pHobj, &pHsub,
             &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
PMQAXP    pExitParms;    /* Exit parameter structure */
PMQAXC    pExitContext;  /* Exit context structure */
PMQHCONN  pHconn;        /* Connection handle */
PPMQSD    ppSubDesc;     /* Subscription descriptor */
PPMQHOBJ  ppHobj;        /* Object Handle */
PPMQHOBJ  ppHsub;        /* Subscription handle */
PMQLONG    pCompCode;    /* Completion code */
PMQLONG    pReason;      /* Reason code qualifying completion code */
```

Subscription request - MQ_SUBRQ_EXIT

MQ_SUBRQ_EXIT provides a subscription request exit function to perform *before* and *after* subscription request processing. Use function identifier MQXF_SUBRQ with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* subscription request call exit functions.

The interface to this function is:

```
MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pHsub, &Action, &pSubRqOpts,
               &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input/output

Connection handle.

pHsub (MQHOBJ) input/output

Subscription handle

Action (MQLONG) input/output

Action

pSubRqOpts (MQSRO) input/output

CompCode (MQLONG) - input/output

Completion code, valid values for which are:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed

Reason (MQLONG) - input/output

Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

MQRC_NONE

(0, x'000') No reason to report.

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;    /* Exit parameter structure */
MQAXC    ExitContext;  /* Exit context structure */
MQHCONN  Hconn;        /* Connection handle */
PMQLONG  pHsub;        /* Subscription handle */
```

```

MQLONG    Action;           /* Action */
PMQSRO    pSubRqOpts;       /* Subscription Request Options */
MQLONG    CompCode;         /* Completion code */
MQLONG    Reason;           /* Reason code qualifying completion code */

```

The queue manager then logically calls the exit as follows:

```

MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pHsub, &Action, &pSubRqOpts,
               &CompCode, &Reason);

```

Your exit must match the following C function prototype:

```

void MQENTRY MQ_SUBRQ_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,          /* Address of connection handle */
PPMHOBJS  ppHsub;           /* Address of Subscription handle */
MQLONG    pAction;          /* Address of Action */
PMQSRO    ppSubRqOpts;      /* Address of Subscription Request Options */
MQLONG    pCompCode;        /* Address of completion code */
MQLONG    pReason;          /* Address of reason code qualifying completion
                             code */

```

xa_close - XA_CLOSE_EXIT

XA_CLOSE_EXIT provides an xa_close exit function to perform before and after xa_close processing. Use function identifier MQXF_XACLOSE with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_close call exit functions.

The interface to this function is:

```

XA_CLOSE_EXIT (&ExitParms, &ExitContext, &Hconn, &pXa_info, &Rmid, &Flags, &XARetCode)

```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXa_info (PMQCHAR) - input/output

Instance-specific resource manager information.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```

MQAXP    ExitParms;         /* Exit parameter structure */
MQAXC    ExitContext;       /* Exit context structure */
MQHCONN  Hconn;             /* Connection handle */
PMQCHAR  pXa_info;          /* Instance-specific RM info */
MQLONG    Rmid;             /* Resource manager identifier */
MQLONG    Flags;            /* Resource manager options */
MQLONG    XARetCode;        /* Response from XA call */

```

The queue manager then logically calls the exit as follows:

```

XA_CLOSE_EXIT (&ExitParms, &ExitContext, &Hconn, &pXa_info, &Rmid, &Flags, &XARetCode);

```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_CLOSE_EXIT (
    PMQAXP    pExitParms,    /* Address of exit parameter structure */
    PMQAXC    pExitContext,  /* Address of exit context structure */
    PMQHCONN  pHconn,        /* Address of connection handle */
    PPMQCHAR  ppXa_info,     /* Address of instance-specific RM info */
    PMQLONG   pRmid,         /* Address of resource manager identifier */
    PMQLONG   pFlags,        /* Address of resource manager options */
    PMQLONG   pXARetCode);   /* Address of response from XA call */
```

xa_commit - XA_COMMIT_EXIT

XA_COMMIT_EXIT provides an xa_commit exit function to perform before and after xa_commit processing. Use function identifier MQXF_XACOMMIT with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_commit call exit functions.

The interface to this function is:

```
XA_COMMIT_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;    /* Exit parameter structure */
MQAXC    ExitContext;  /* Exit context structure */
MQHCONN  Hconn;        /* Connection handle */
MQPTR    pXID;         /* Transaction branch ID */
MQLONG   Rmid;         /* Resource manager identifier */
MQLONG   Flags;        /* Resource manager options */
MQLONG   XARetCode;    /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_COMMIT_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_COMMIT_EXIT (
    PMQAXP    pExitParms,    /* Address of exit parameter structure */
    PMQAXC    pExitContext,  /* Address of exit context structure */
    PMQHCONN  pHconn,        /* Address of connection handle */
    MQPTR     ppXID,         /* Address of transaction branch ID */
    PMQLONG   pRmid,        /* Address of resource manager identifier */
    PMQLONG   pFlags,        /* Address of resource manager options */
    PMQLONG   pXARetCode);   /* Address of response from XA call */
```

xa_complete - XA_COMPLETE_EXIT

XA_COMPLETE_EXIT provides an xa_complete exit function to perform before and after xa_complete processing. Use function identifier MQXF_XACOMPLETE with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_complete call exit functions.

The interface to this function is:

```
XA_COMPLETE_EXIT (&ExitParms, &ExitContext, &Hconn, &pHandle, &pRetVal, &Rmid, &Flags,
&XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pHandle (PMQLONG) - input/output

Pointer to asynchronous operation.

pRetVal (PMQLONG) - input/output

Return value of asynchronous operation.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP   ExitParms;    /* Exit parameter structure */
MQAXC   ExitContext;  /* Exit context structure */
MQHCONN Hconn;        /* Connection handle */
PMQLONG pHandle;      /* Ptr to asynchronous op */
PMQLONG pRetVal;      /* Return value of async op */
MQLONG  Rmid;         /* Resource manager identifier */
MQLONG  Flags;        /* Resource manager options*/
MQLONG  XARetCode;    /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_COMPLETE_EXIT (&ExitParms, &ExitContext, &Hconn, &pHandle, &pRetVal, &Rmid, &Flags,
&XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_COMPLETE_EXIT (
    PMQAXP   pExitParms,    /* Address of exit parameter structure */
    PMQAXC   pExitContext,  /* Address of exit context structure */
    PMQHCONN pHconn,        /* Address of connection handle */
    PPMQLONG ppHandle,      /* Address of ptr to asynchronous op */
    PPMQLONG ppRetVal,      /* Address of return value of async op */
    PMQLONG  pRmid,         /* Address of resource manager identifier */
    PMQLONG  pFlags,        /* Address of resource manager options*/
    PMQLONG  pXARetCode);   /* Address of response from XA call */
```

xa_end - XA_END_EXIT

XA_END_EXIT provides an `xa_end` exit function to perform before and after `xa_end` processing. Use function identifier `MQXF_XAEND` with exit reasons `MQXR_BEFORE` and `MQXR_AFTER` to register the before and after `xa_end` call exit functions.

The interface to this function is:

```
XA_END_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP   ExitParms;    /* Exit parameter structure */
MQAXC   ExitContext;  /* Exit context structure */
MQHCONN Hconn;        /* Connection handle */
MQPTR   pXID;         /* Transaction branch ID */
MQLONG  Rmid;         /* Resource manager identifier */
MQLONG  Flags;        /* Resource manager options */
MQLONG  XARetCode;    /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_END_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_END_EXIT (
    PMQAXP pExitParms, /* Address of exit parameter structure */
    PMQAXC pExitContext, /* Address of exit context structure */
    PMQHCONN pHconn, /* Address of connection handle */
    PMQPTR ppXID, /* Address of transaction branch ID */
    PMQLONG pRmid, /* Address of resource manager identifier */
    PMQLONG pFlags, /* Address of resource manager options */
    PMQLONG pXARetCode); /* Address of response from XA call */
```

xa_forget - XA_FORGET_EXIT

XA_FORGET_EXIT provides an `xa_forget` exit function to perform before and after `xa_forget` processing. Use function identifier `MQXF_XAFORGET` with exit reasons `MQXR_BEFORE` and `MQXR_AFTER` to register the before and after `xa_forget` call exit functions.

The interface to this function is:

```
XA_FORGET_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```

MQAXP  ExitParms;    /* Exit parameter structure */
MQAXC  ExitContext;  /* Exit context structure */
MQHCONN Hconn;       /* Connection handle */
MQPTR  pXID;         /* Transaction branch ID */
MQLONG Rmid;         /* Resource manager identifier */
MQLONG Flags;        /* Resource manager options */
MQLONG XARetCode;    /* Response from XA call */

```

The queue manager then logically calls the exit as follows:

```
XA_FORGET_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```

typedef void MQENTRY XA_FORGET_EXIT (
    PMQAXP  pExitParms,    /* Address of exit parameter structure */
    PMQAXC  pExitContext,  /* Address of exit context structure */
    PMQHCONN pHconn,       /* Address of connection handle */
    PMQPTR  ppXID,         /* Address of transaction branch ID */
    PMQLONG pRmid,         /* Address of resource manager identifier */
    PMQLONG pFlags,        /* Address of resource manager options */
    PMQLONG pXARetCode);   /* Address of response from XA call */

```

xa_open - XA_OPEN_EXIT

XA_OPEN_EXIT provides an `xa_open` exit function to perform before and after `xa_open` processing. Use function identifier `MQXF_XAOPEN` with exit reasons `MQXR_BEFORE` and `MQXR_AFTER` to register the before and after `xa_open` call exit functions.

The interface to this function is:

```
XA_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pXa_info, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXa_info (PMQCHAR) - input/output

Instance-specific resource manager information.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP  ExitParms;    /* Exit parameter structure */
MQAXC  ExitContext; /* Exit context structure */
MQHCONN Hconn;      /* Connection handle */
PMQCHAR pXa_info;   /* Instance-specific RM info */
MQLONG Rmid;        /* Resource manager identifier */
MQLONG Flags;       /* Resource manager options */
MQLONG XARetCode;   /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pXa_info, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_OPEN_EXIT (
    PMQAXP  pExitParms, /* Address of exit parameter structure */
    PMQAXC  pExitContext, /* Address of exit context structure */
    PMQHCONN pHconn, /* Address of connection handle */
    PPMQCHAR ppXa_info, /* Address of instance-specific RM info */
    PMQLONG pRmid, /* Address of resource manager identifier */
    PMQLONG pFlags, /* Address of resource manager options */
    PMQLONG pXARetCode); /* Address of response from XA call */
```

xa_prepare - XA_PREPARE_EXIT

XA_PREPARE_EXIT provides an xa_prepare exit function to perform before and after xa_prepare processing. Use function identifier MQXF_XAPREPARE with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_prepare call exit functions.

The interface to this function is:

```
XA_PREPARE_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP  ExitParms; /* Exit parameter structure */
MQAXC  ExitContext; /* Exit context structure */
MQHCONN Hconn; /* Connection handle */
MQPTR  pXID; /* Transaction branch ID */
MQLONG Rmid; /* Resource manager identifier */
MQLONG Flags; /* Resource manager options */
MQLONG XARetCode; /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_PREPARE_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_PREPARE_EXIT (
    PMQAXP  pExitParms, /* Address of exit parameter structure */
    PMQAXC  pExitContext, /* Address of exit context structure */
    PMQHCONN pHconn, /* Address of connection handle */
    PMQPTR  ppXID, /* Address of transaction branch ID */
    PMQLONG pRmid, /* Address of resource manager identifier */
    PMQLONG pFlags, /* Address of resource manager options */
    PMQLONG pXARetCode); /* Address of response from XA call */
```

xa_recover - XA_RECOVER_EXIT

XA_RECOVER_EXIT provides an `xa_recover` exit function to perform before and after `xa_recover` processing. Use function identifier `MQXF_XARECOVER` with exit reasons `MQXR_BEFORE` and `MQXR_AFTER` to register the before and after `xa_recover` call exit functions.

The interface to this function is:

```
XA_RECOVER_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Count, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Count (MQLONG) - input/output

Maximum XIDs in XID array

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP  ExitParms; /* Exit parameter structure */
MQAXC  ExitContext; /* Exit context structure */
MQHCONN Hconn; /* Connection handle */
MQPTR  pXID; /* Transaction branch ID */
```

```

MQLONG Count;      /* Max XIDs in XID array */
MQLONG Rmid;        /* Resource manager identifier */
MQLONG Flags;       /* Resource manager options*/
MQLONG XARetCode;   /* Response from XA call */

```

The queue manager then logically calls the exit as follows:

```
XA_RECOVER_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Count, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```

typedef void MQENTRY XA_RECOVER_EXIT (
    PMQAXP  pExitParms, /* Address of exit parameter structure */
    PMQAXC  pExitContext, /* Address of exit context structure */
    PMQHCONN pHconn, /* Address of connection handle */
    PMQPTR  ppXID, /* Address of transaction branch ID */
    PMQLONG pCount, /* Address of max XIDs in XID array */
    PMQLONG pRmid, /* Address of resource manager identifier */
    PMQLONG pFlags, /* Address of resource manager options*/
    PMQLONG pXARetCode); /* Address of response from XA call */

```

xa_rollback - XA_ROLLBACK_EXIT

XA_ROLLBACK_EXIT provides an xa_rollback exit function to perform before and after xa_rollback processing. Use function identifier MQXF_XAROLLBACK with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_rollback call exit functions.

The interface to this function is:

```
XA_ROLLBACK_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```

MQAXP  ExitParms; /* Exit parameter structure */
MQAXC  ExitContext; /* Exit context structure */
MQHCONN Hconn; /* Connection handle */
MQPTR  pXID; /* Transaction branch ID */
MQLONG Rmid; /* Resource manager identifier */
MQLONG Flags; /* Resource manager options*/
MQLONG XARetCode; /* Response from XA call */

```

The queue manager then logically calls the exit as follows:

```
XA_ROLLBACK_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_ROLLBACK_EXIT (
    PMQAXP    pExitParms, /* Address of exit parameter structure */
    PMQAXC    pExitContext, /* Address of exit context structure */
    PMQHCONN  pHconn, /* Address of connection handle */
    PMQPTR    ppXID, /* Address of transaction branch ID */
    PMQLONG   pRmid, /* Address of resource manager identifier */
    PMQLONG   pFlags, /* Address of resource manager options */
    PMQLONG   pXARetCode); /* Address of response from XA call */
```

xa_start - XA_START_EXIT

XA_START_EXIT provides an xa_start exit function to perform before and after xa_start processing. Use function identifier MQXF_XASTART with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after xa_start call exit functions.

The interface to this function is:

```
XA_START_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP    ExitParms; /* Exit parameter structure */
MQAXC    ExitContext; /* Exit context structure */
MQHCONN  Hconn; /* Connection handle */
MQPTR    pXID; /* Transaction branch ID */
MQLONG   Rmid; /* Resource manager identifier */
MQLONG   Flags; /* Resource manager options */
MQLONG   XARetCode; /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
XA_START_EXIT (&ExitParms, &ExitContext, &Hconn, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY XA_START_EXIT (
    PMQAXP    pExitParms, /* Address of exit parameter structure */
    PMQAXC    pExitContext, /* Address of exit context structure */
    PMQHCONN  pHconn, /* Address of connection handle */
    PMQPTR    ppXID, /* Address of transaction branch ID */
    PMQLONG   pRmid, /* Address of resource manager identifier */
    PMQLONG   pFlags, /* Address of resource manager options */
    PMQLONG   pXARetCode); /* Address of response from XA call */
```

ax_reg - AX_REG_EXIT

AX_REG_EXIT provides an ax_reg exit function to perform before and after ax_reg processing. Use function identifier MQXF_AXREG with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after ax_reg call exit functions.

The interface to this function is:

```
AX_REG_EXIT (&ExitParms, &ExitContext, &pXID, &Rmid, &Flags, &XARetCode)
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Hconn (MQHCONN) - input

Connection handle.

pXID (MQPTR) - input/output

Transaction branch ID.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```
MQAXP  ExitParms;    /* Exit parameter structure */
MQAXC  ExitContext;  /* Exit context structure */
MQPTR  pXID;         /* Transaction branch ID */
MQLONG Rmid;         /* Resource manager identifier */
MQLONG Flags;        /* Resource manager options */
MQLONG XARetCode;    /* Response from XA call */
```

The queue manager then logically calls the exit as follows:

```
AX_REG_EXIT (&ExitParms, &ExitContext, &pXID, &Rmid, &Flags, &XARetCode);
```

Your exit must match the following C function prototype:

```
typedef void MQENTRY AX_REG_EXIT (
    PMQAXP pExitParms, /* Address of exit parameter structure */
    PMQAXC pExitContext, /* Address of exit context structure */
    PMQPTR ppXID, /* Address of transaction branch ID */
    PMQLONG pRmid, /* Address of resource manager identifier */
    PMQLONG pFlags, /* Address of resource manager options */
    PMQLONG pXARetCode); /* Address of response from XA call */
```

ax_unreg - AX_UNREG_EXIT

AX_UNREG_EXIT provides an ax_unreg exit function to perform before and after ax_unreg processing. Use function identifier MQXF_AXUNREG with exit reasons MQXR_BEFORE and MQXR_AFTER to register the before and after ax_unreg call exit functions.

The interface to this function is:

```
AX_UNREG_EXIT (&ExitParms, &ExitContext, &Rmid, &Flags, &XARetCode);
```

where the parameters are:

ExitParms (MQAXP) - input/output

Exit parameter structure.

ExitContext (MQAXC) - input/output

Exit context structure.

Rmid (MQLONG) - input/output

Resource manager identifier.

Flags (MQLONG) - input/output

Resource manager options.

XARetCode (MQLONG) - input/output

Response from XA call.

C language invocation

The queue manager logically defines the following variables:

```

MQAXP  ExitParms;    /* Exit parameter structure */
MQAXC  ExitContext;  /* Exit context structure */
MQLONG Rmid;         /* Resource manager identifier */
MQLONG Flags;        /* Resource manager options*/
MQLONG XARetCode;    /* Response from XA call */

```

The queue manager then logically calls the exit as follows:

```

AX_UNREG_EXIT (&ExitParms, &ExitContext, &Rmid, &Flags, &XARetCode);

```

Your exit must match the following C function prototype:

```

typedef void MQENTRY AX_UNREG_EXIT (
    PMQAXP pExitParms,    /* Address of exit parameter structure */
    PMQAXC pExitContext,  /* Address of exit context structure */
    PMQLONG pRmid,        /* Address of resource manager identifier */
    PMQLONG pFlags,       /* Address of resource manager options*/
    PMQLONG pXARetCode);  /* Address of response from XA call */

```

General information on invoking exit functions

This topic provides some general guidance to help you to plan your exits, particularly related to handling errors and unexpected events.

Exit failure

If an exit function abnormally terminates after a destructive, out of syncpoint, MQGET call but before the message has been passed to the application, the exit handler can recover from the failure, and pass control to the application.

In this case, the message might be lost. This is like what happens when an application fails immediately after receiving a message from a queue.

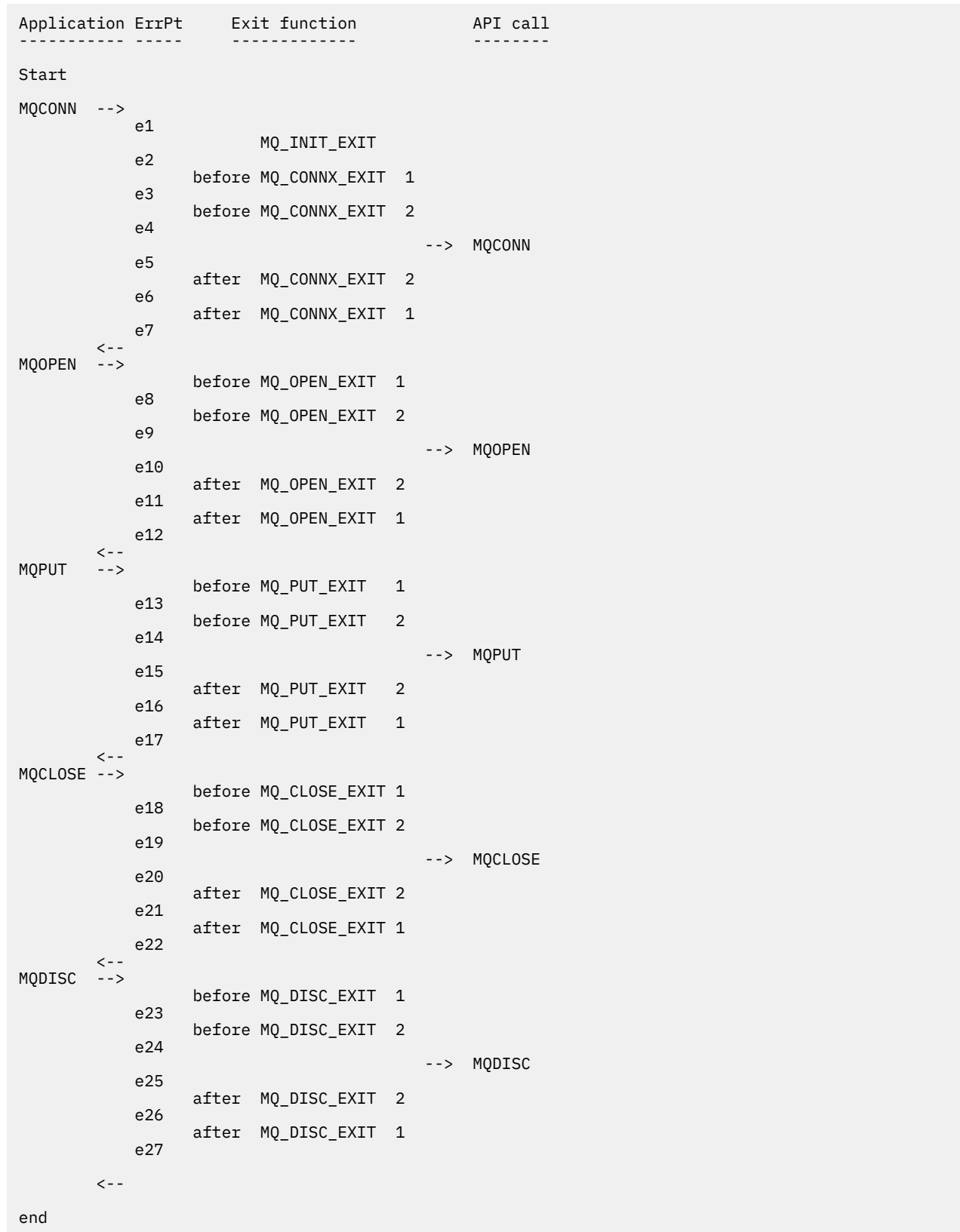
The MQGET call might complete with MQCC_FAILED and MQRC_API_EXIT_ERROR.

If a *before* API call exit function terminates abnormally, the exit handler can recover from the failure and pass control to the application without processing the API call. In this event, the exit function must recover any resources that it owns.

If chained exits are in use, the *after* API call exits for any *before* API call exits that had successfully been driven can themselves be driven. The API call might fail with MQCC_FAILED and MQRC_API_EXIT_ERROR.

Example error handling for exit functions

The following diagram shows the points (e N) at which errors can occur. It is only an example to show how exits behave and should be read together with the following table. In this example, two exit functions are invoked both before and after each API call to show the behavior with chained exits.



The following table lists the actions to be taken at each error point. Only a subset of the error points have been covered, as the rules shown here can apply to all others. It is the actions that specify the intended behavior in each case.

Table 657. API exit errors and appropriate actions to take		
Err Pt	Description	Actions
e1	Error while setting up environment setup.	<ol style="list-style-type: none"> 1. Undo environment setup as required 2. Drive no exit functions 3. Fail MQCONN with MQCC_FAILED, MQRC_API_EXIT_LOAD_ERROR
e2	MQ_INIT_EXIT function completes with: <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Clean up environment 2. Fail MQCONN with MQCC_FAILED, MQRC_API_EXIT_INIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Clean up environment
e3	<i>Before</i> MQ_CONNX_EXIT 1 function completes with: <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive MQ_TERM_EXIT function 2. Clean up environment 3. Fail MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Drive MQ_TERM_EXIT function if required 3. Clean up environment if required
e4	<i>Before</i> MQ_CONNX_EXIT 2 function completes with: <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive <i>after</i> MQ_CONNX_EXIT 1 function 2. Drive MQ_TERM_EXIT function 3. Clean up environment 4. Fail MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Drive <i>after</i> MQ_CONNX_EXIT 1 function if exit not suppressed 3. Drive MQ_TERM_EXIT function if required 4. Clean up environment if required

Table 657. API exit errors and appropriate actions to take (continued)

Err Pt	Description	Actions
e5	MQCONN call fails.	<ol style="list-style-type: none"> 1. Pass MQCONN CompCode and Reason 2. Drive <i>after</i> MQ_CONNX_EXIT 2 function if the <i>before</i> MQ_CONNX_EXIT 2 succeeded and the exit is not suppressed 3. Drive <i>after</i> MQ_CONNX_EXIT 1 function if the <i>before</i> MQ_CONNX_EXIT 1 succeeded and the exit is not suppressed 4. Drive MQ_TERM_EXIT function 5. Clean up environment
e6	<p><i>After</i> MQ_CONNX_EXIT 2 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive <i>after</i> MQ_CONNX_EXIT 1 function 2. Complete MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Drive <i>after</i> MQ_CONNX_EXIT 1 function if required
e7	<p><i>After</i> MQ_CONNX_EXIT 1 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED, complete MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*¹
e8	<p><i>Before</i> MQ_OPEN_EXIT 1 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED, complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*¹
e9	<p><i>Before</i> MQ_OPEN_EXIT 2 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive <i>after</i> MQ_OPEN_EXIT 1 function 2. Complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*¹
e10	MQOPEN call fails	<ol style="list-style-type: none"> 1. Pass MQOPEN CompCode and Reason 2. Drive <i>after</i> MQ_OPEN_EXIT 2 function if exit not suppressed 3. Drive <i>after</i> MQ_OPEN_EXIT 1 function if exit not suppressed and if chained exits not suppressed

Table 657. API exit errors and appropriate actions to take (continued)		
Err Pt	Description	Actions
e1 1	<p>After MQ_OPEN_EXIT 2 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive <i>after</i> MQ_OPEN_EXIT 1 function 2. Complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Drive <i>after</i> MQ_OPEN_EXIT 1 function if exit not suppressed
e2 5	<p>After MQ_DISC_EXIT 2 function completes with:</p> <ul style="list-style-type: none"> • MQXCC_FAILED • MQXCC_* 	<ul style="list-style-type: none"> • For MQXCC_FAILED: <ol style="list-style-type: none"> 1. Drive <i>after</i> MQ_DISC_EXIT 1 function 2. Drive MQ_TERM_EXIT function 3. Clean up exit execution environment 4. Complete MQDISC call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* <ol style="list-style-type: none"> 1. Act as for the values of MQXCC_* and MQXR2_*¹ 2. Drive MQ_TERM_EXIT function 3. Clean up exit execution environment

Note:

1. The values of MQXCC_* and MQXR2_* and their corresponding actions are defined in [How queue managers process exit functions](#).

ExitResponse fields set incorrectly

This topic gives information about what would happen when the ExitResponse field is set to anything but the supported values.

If the ExitResponse field is set to a value other than one of the supported values, the following actions apply:

- For a *before* MQCONN or MQDISC API exit function:
 - The ExitResponse2 value is ignored.
 - No further *before* exit functions in the exit chain (if any) are invoked; the API call itself is not issued.
 - For any *before* exits that were successfully called, the *after* exits are called in reverse order.
 - If registered, the termination exit functions for those *before* MQCONN or MQDISC exit functions in the chain that were successfully invoked are driven to clean up after these exit functions.
 - The MQCONN or MQDISC call fails with MQRC_API_EXIT_ERROR.
- For a *before* IBM MQ API exit function other than MQCONN or MQDISC:
 - The ExitResponse2 value is ignored.
 - No further *before* or *after* data conversion functions in the exit chain (if any) are invoked.
 - For any *before* exits that were successfully called, the *after* exits are called in reverse order.
 - The IBM MQ API call itself is not issued.
 - The IBM MQ API call fails with MQRC_API_EXIT_ERROR.

- For an *after* MQCONN or MQDISC API exit function:
 - The ExitResponse2 value is ignored.
 - The remaining exit functions that were successfully called before the API call are called in reverse order.
 - If registered, the termination exit functions for those *before* or *after* MQCONN or MQDISC exit functions in the chain that were successfully invoked are driven to clean up after the exit.
 - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
 - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
 - The IBM MQ API call is successfully issued.
- For an *after* IBM MQ API call exit function other than MQCONN or MQDISC:
 - The ExitResponse2 value is ignored.
 - The remaining exit functions that were successfully called before the API call are called in reverse order.
 - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
 - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
 - The IBM MQ API call is successfully issued.
- For the *before* data conversion on get exit function:
 - The ExitResponse2 value is ignored.
 - The remaining exit functions that were successfully called before the API call are called in reverse order.
 - The message is not converted, and the unconverted message is returned to the application.
 - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
 - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
 - The IBM MQ API call is successfully issued.

Note: As the error is with the exit, it is better to return MQRC_API_EXIT_ERROR than to return MQRC_NOT_CONVERTED.

If an exit function sets the ExitResponse2 field to a value other than one of the supported values, a value of MQXR2_DEFAULT_CONTINUATION is assumed instead.

Installable services interface reference information

This collection of topics provides reference information for the installable services.

The functions and data types are listed in alphabetical order within the group for each service type.

How the functions are shown

How the installable services functions are documented.

For each function there is a description, including the function identifier (for MQZEP).

The *parameters* are shown listed in the order they must occur. They must all be present.

Each parameter name is followed by its data type. These are the elementary data types described in the [“Elementary data types” on page 231](#).

The C language invocation is also given, after the description of the parameters.

MQZ_AUTHENTICATE_USER - Authenticate user

This function is provided by an MQZAS_VERSION_5 authorization service component, and is invoked by the queue manager to authenticate a user, or to set identity context fields. It is invoked when the IBM MQ user application context is established.

The application context is established during connect calls at the point where the application's user context is initialized, and at each point where the application's user context is changed. Each time a connect call is made, the application's user context information is reacquired in the *IdentityContext* field.

The function identifier for this function (for MQZEP) is MQZID_AUTHENTICATE_USER.

Syntax

MQZ_AUTHENTICATE_USER (*QMgrName* , *SecurityParms* , *ApplicationContext* , *IdentityContext* , *CorrelationPtr* , *ComponentData* , *Continuation* , *CompCode* , *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

SecurityParms

Type: MQCSP - input

Security parameters. Data relating to the user ID, password, and authentication type. If the AuthenticationType attribute of the MQCSP structure is specified as MQCSP_AUTH_USER_ID_AND_PWD, both the user ID and password are compared against the equivalent fields in the IdentityContext (MQZIC) parameter to determine whether they match. For more information, see “MQCSP - Security parameters” on page 320.

During an MQCONN MQI call this parameter contains null, or default values.

ApplicationContext

Type: MQZAC - input

Application context. Data relating to the calling application. See [MQZAC - Application context](#) for details.

During every MQCONN or MQCONNX MQI call, the user context information in the MQZAC structure is reacquired.

IdentityContext

Type: MQZIC - input/output

Identity context. On input to the authenticate user function, this identifies the current identity context. The authenticate user function can change this, at which point the queue manager adopts the new identity context. See [MQZIC - Identity context](#) for more details on the MQZIC structure.

CorrelationPtr

Type: MQPTR - output

Correlation pointer. Specifies the address of any correlation data. This pointer is subsequently passed on to other OAM calls.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the `ComponentDataLength` parameter of the `MQZ_INIT_AUTHORITY` call.

Continuation

Type: `MLONG` - output

Continuation flag. You can specify the following values:

MQZCI_DEFAULT

Continuation dependent on other components.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: `MLONG` - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: `MLONG` - output

Reason code qualifying *CompCode*.

If *CompCode* is `MQCC_OK`:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is `MQCC_FAILED`:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information about these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_AUTHENTICATE_USER (QMgrName, SecurityParms, ApplicationContext,  
                      IdentityContext, &CorrelationPtr, ComponentData,  
                      &Continuation, &CompCode, &Reason);
```

Declare the parameters passed to the service as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCSP     SecurityParms;      /* Security parameters */  
MQZAC     ApplicationContext; /* Application context */  
MQZIC     IdentityContext;    /* Identity context */  
MQPTR     CorrelationPtr;     /* Correlation pointer */  
MQBYTE    ComponentData[n];   /* Component data */  
MLONG     Continuation;       /* Continuation indicator set by  
                               component */  
MLONG     CompCode;           /* Completion code */  
MLONG     Reason;            /* Reason code qualifying CompCode */
```

MQZ_CHECK_AUTHORITY - Check authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is started by the queue manager to check whether an entity has authority to perform a particular action, or actions, on a specified object.

The function identifier for this function (for MQZEP) is MQZID_CHECK_AUTHORITY.

Syntax

MQZ_CHECK_AUTHORITY(*QMgrName* , *EntityName* , *EntityType* , *ObjectName* ,
ObjectType , *Authority* , *ComponentData* , *Continuation* , *CompCode* , *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityName

Type: MQCHAR12 - input

Entity name. The name of the entity whose authorization to the object is to be checked. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

It is not essential for this entity to be known to the underlying security service. If it is not known, the authorizations of the special **nobody** group (to which all entities are assumed to belong) are used for the check. An all-blank name is valid and can be used in this way.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by EntityName. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object to which access is required. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority

Type: MQLONG - input

Authority to be checked. If one authorization is being checked, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authorization is being checked, it is the bitwise OR of the corresponding MQZAO_* constants.

The following authorizations apply to use of the MQI calls:

MQZAO_CONNECT

Ability to use the MQCONN call.

MQZAO_BROWSE

Ability to use the MQGET call with a browse option.

This allows the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_MSG_UNDER_CURSOR, or MQGMO_BROWSE_NEXT option to be specified on the MQGET call.

MQZAO_INPUT

Principal. Ability to use the MQGET call with an input option.

This allows the MQOO_INPUT_SHARED, MQOO_INPUT_EXCLUSIVE, or MQOO_INPUT_AS_Q_DEF option to be specified on the MQOPEN call.

MQZAO_OUTPUT

Ability to use the MQPUT call.

This allows the MQOO_OUTPUT option to be specified on the MQOPEN call.

MQZAO_INQUIRE

Ability to use the MQINQ call.

This allows the MQOO_INQUIRE option to be specified on the MQOPEN call.

MQZAO_SET

Ability to use the MQSET call.

This allows the MQOO_SET option to be specified on the MQOPEN call.

MQZAO_PASS_IDENTITY_CONTEXT

Ability to pass identity context.

This allows the MQOO_PASS_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_PASS_ALL_CONTEXT

Ability to pass all context.

This allows the MQOO_PASS_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_IDENTITY_CONTEXT

Ability to set identity context.

This allows the MQOO_SET_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_ALL_CONTEXT

Ability to set all context.

This allows the MQOO_SET_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_ALTERNATE_USER_AUTHORITY

Ability to use alternate user authority.

This allows the MQOO_ALTERNATE_USER_AUTHORITY option to be specified on the MQOPEN call, and the MQPMO_ALTERNATE_USER_AUTHORITY option to be specified on the MQPUT1 call.

MQZAO_ALL_MQI

All of the MQI authorizations.

This enables all of the authorizations.

The following authorizations apply to administration of a queue manager:

MQZAO_CREATE

Ability to create objects of a specified type.

MQZAO_DELETE

Ability to delete a specified object.

MQZAO_DISPLAY

Ability to display the attributes of a specified object.

MQZAO_CHANGE

Ability to change the attributes of a specified object.

MQZAO_CLEAR

Ability to delete all messages from a specified queue.

MQZAO_AUTHORIZE

Ability to authorize other users for a specified object.

MQZAO_CONTROL

Ability to start or stop a listener, service, or non-client channel object, and the ability to ping a non-client channel object.

MQZAO_CONTROL_EXTENDED

Ability to reset a sequence number, or resolve an indoubt message on a non-client channel object.

MQZAO_ALL_ADMIN

Ability to set identity context.

All of the administration authorizations, other than MQZAO_CREATE.

The following authorizations apply to both use of the MQI and to administration of a queue manager:

MQZAO_ALL

All authorizations, other than MQZAO_CREATE.

MQZAO_NONE

No authorizations.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

If the call to a component fails (that is, *CompCode* returns MQCC_FAILED), and the *Continuation* parameter is MQZCI_DEFAULT or MQZCI_CONTINUE, the queue manager continues to call other components if there are any.

If the call succeeds (that is, *CompCode* returns MQCC_OK) no other components are called no matter what the setting of *Continuation* is.

If the call fails and the *Continuation* parameter is MQZCI_STOP then no other components are called and the error is returned to the queue manager. Components have no knowledge of previous calls, so the *Continuation* parameter is always set to MQZCI_DEFAULT before the call.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_CHECK_AUTHORITY (QMGrName, EntityName, EntityType, ObjectName,  
                    ObjectType, Authority, ComponentData,  
                    &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```

MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR12  EntityName;         /* Entity name */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority to be checked */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */

```

MQZ_CHECK_AUTHORITY_2 - Check authority (extended)

This function is provided by a MQZAS_VERSION_2 authorization service component, and is started by the queue manager to check whether an entity has authority to perform a particular action, or actions, on a specified object.

The function identifier for this function (for MQZEP) is MQZID_CHECK_AUTHORITY.

MQZ_CHECK_AUTHORITY_2 is like MQZ_CHECK_AUTHORITY, but with the *EntityName* parameter replaced by the *EntityData* parameter.

Syntax

MQZ_CHECK_AUTHORITY_2(*QMgrName* , *EntityData* , *EntityType* , *ObjectName* , *ObjectType* , *Authority* , *ComponentData* , *Continuation* , *CompCode* , *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity with authorization to the object that is to be checked. See [“MQZED - Entity descriptor” on page 1681](#) for details.

It is not essential for this entity to be known to the underlying security service. If it is not known, the authorizations of the special **nobody** group (to which all entities are assumed to belong) are used for the check. An all-blank name is valid and can be used in this way.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityData*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object to which access is required. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority to be checked. If one authorization is being checked, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authorization is being checked, it is the bitwise OR of the corresponding MQZAO_* constants.

The following authorizations apply to use of the MQI calls:

MQZAO_CONNECT

Ability to use the MQCONN call.

MQZAO_BROWSE

Ability to use the MQGET call with a browse option.

This allows the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_MSG_UNDER_CURSOR, or MQGMO_BROWSE_NEXT option to be specified on the MQGET call.

MQZAO_INPUT

Principal. Ability to use the MQGET call with an input option.

This allows the MQOO_INPUT_SHARED, MQOO_INPUT_EXCLUSIVE, or MQOO_INPUT_AS_Q_DEF option to be specified on the MQOPEN call.

MQZAO_OUTPUT

Ability to use the MQPUT call.

This allows the MQOO_OUTPUT option to be specified on the MQOPEN call.

MQZAO_INQUIRE

Ability to use the MQINQ call.

This allows the MQOO_INQUIRE option to be specified on the MQOPEN call.

MQZAO_SET

Ability to use the MQSET call.

This allows the MQOO_SET option to be specified on the MQOPEN call.

MQZAO_PASS_IDENTITY_CONTEXT

Ability to pass identity context.

This allows the MQOO_PASS_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_PASS_ALL_CONTEXT

Ability to pass all context.

This allows the MQOO_PASS_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_IDENTITY_CONTEXT

Ability to set identity context.

This allows the MQOO_SET_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_ALL_CONTEXT

Ability to set all context.

This allows the MQOO_SET_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_ALTERNATE_USER_AUTHORITY

Ability to use alternate user authority.

This allows the MQOO_ALTERNATE_USER_AUTHORITY option to be specified on the MQOPEN call, and the MQPMO_ALTERNATE_USER_AUTHORITY option to be specified on the MQPUT1 call.

MQZAO_ALL_MQI

All of the MQI authorizations.

This enables all of the authorizations.

The following authorizations apply to administration of a queue manager:

MQZAO_CREATE

Ability to create objects of a specified type.

MQZAO_DELETE

Ability to delete a specified object.

MQZAO_DISPLAY

Ability to display the attributes of a specified object.

MQZAO_CHANGE

Ability to change the attributes of a specified object.

MQZAO_CLEAR

Ability to delete all messages from a specified queue.

MQZAO_AUTHORIZE

Ability to authorize other users for a specified object.

MQZAO_CONTROL

Ability to start or stop a listener, service, or non-client channel object, and the ability to ping a non-client channel object.

MQZAO_CONTROL_EXTENDED

Ability to reset a sequence number, or resolve an indoubt message on a non-client channel object.

MQZAO_ALL_ADMIN

Ability to set identity context.

All of the administration authorizations, other than MQZAO_CREATE.

The following authorizations apply to both use of the MQI and to administration of a queue manager:

MQZAO_ALL

All authorizations, other than MQZAO_CREATE.

MQZAO_NONE

No authorizations.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_CHECK_AUTHORITY_2 (QMgrName, &EntityData, EntityType,  
                        ObjectName, ObjectType, Authority, ComponentData,  
                        &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQZED     EntityData;         /* Entity data */  
MQLONG    EntityType;         /* Entity type */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQLONG    Authority;          /* Authority to be checked */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_CHECK_PRIVILEGED - Check if user is privileged

This function is provided by an MQZAS_VERSION_6 authorization service component, and is invoked by the queue manager to determine whether a specified user is a privileged user.

The function identifier for this function (for MQZEP) is MQZID_CHECK_PRIVILEGED.

Syntax

```
MQZ_CHECK_PRIVILEGED( QMgrName , EntityData , EntityType , ComponentData ,  
                     Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity that is to be checked. For more information, see [“MQZED - Entity descriptor”](#) on page 1681.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by EntityData. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

If the call to a component fails (that is, *CompCode* returns MQCC_FAILED), and the *Continuation* parameter is MQZCI_DEFAULT or MQZCI_CONTINUE, the queue manager continues to call other components if there are any.

If the call succeeds (that is, *CompCode* returns MQCC_OK) no other components are called no matter what the setting of *Continuation* is.

If the call fails and the *Continuation* parameter is MQZCI_STOP then no other components are called and the error is returned to the queue manager. Components have no knowledge of previous calls, so the *Continuation* parameter is always set to MQZCI_DEFAULT before the call.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_PRIVILEGED

(2584, X'A18') This user is not a privileged user ID.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_CHECK_PRIVILEGED (QMgrName, &EntityData, EntityType,  
                     ComponentData, &Continuation,  
                     &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQZED     EntityData;         /* Entity name */  
MQLONG    EntityType;         /* Entity type */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_COPY_ALL_AUTHORITY - Copy all authority

This function is provided by an authorization service component. It is started by the queue manager to copy all of the authorizations that are currently in force for a reference object to another object.

The function identifier for this function (for MQZEP) is MQZID_COPY_ALL_AUTHORITY.

Syntax

```
MQZ_COPY_ALL_AUTHORITY( QMgrName , RefObjectName , ObjectName , ObjectType ,  
                        ComponentData , Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

RefObjectName

Type: MQCHAR48 - input

Reference object name. The name of the reference object, the authorizations for which are to be copied. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object for which accesses are to be set. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *RefObjectName* and *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_REF_OBJECT

(2294, X'8F6') Reference object unknown.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_COPY_ALL_AUTHORITY (QMgrName, RefObjectName, ObjectName, ObjectType,  
                        ComponentData, &Continuation, &CompCode,  
                        &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR48  RefObjectName;      /* Reference object name */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_DELETE_AUTHORITY - Delete authority

This function is provided by an authorization service component, and is started by the queue manager to delete all of the authorizations associated with the specified object.

The function identifier for this function (for MQZEP) is MQZID_DELETE_AUTHORITY.

Syntax

```
MQZ_DELETE_AUTHORITY( QMgrName , ObjectName , ObjectType , ComponentData ,  
Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object for which accesses are to be deleted. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_DELETE_AUTHORITY (QMgrName, ObjectName, ObjectType, ComponentData,  
                      &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_ENUMERATE_AUTHORITY_DATA - Enumerate authority data

This function is provided by an MQZAS_VERSION_4 authorization service component, and is started repeatedly by the queue manager to retrieve all of the authority data that matches the selection criteria specified on the first invocation.

The function identifier for this function (for MQZEP) is MQZID_ENUMERATE_AUTHORITY_DATA.

Syntax

```
MQZ_ENUMERATE_AUTHORITY_DATA( QMgrName , StartEnumeration , Filter ,  
AuthorityBufferLength , AuthorityBuffer , AuthorityDataLength , ComponentData ,  
Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

StartEnumeration

Type: MQLONG - input

Flag indicating whether call can start enumeration. This indicates whether the call can start the enumeration of authority data, or continue the enumeration of authority data started by a previous call to MQZ_ENUMERATE_AUTHORITY_DATA. The value is one of the following values:

MQZSE_START

Start enumeration. The call is started with this value to start the enumeration of authority data. The *Filter* parameter specifies the selection criteria to be used to select the authority data returned by this and successive calls.

MQZSE_CONTINUE

Continue enumeration. The call is started with this value to continue the enumeration of authority data. The *Filter* parameter is ignored in this case, and can be specified as the null pointer (the selection criteria are determined by the *Filter* parameter specified by the call that had *StartEnumeration* set to MQZSE_START).

Filter

Type: MQZAD - input

Filter. If *StartEnumeration* is MQZSE_START, *Filter* specifies the selection criteria to be used to select the authority data to return. If *Filter* is the null pointer, no selection criteria are used, that is, all authority data is returned. See “MQZAD - Authority data” on page 1678 for details of the selection criteria that can be used.

If *StartEnumeration* is MQZSE_CONTINUE, *Filter* is ignored, and can be specified as the null pointer.

AuthorityBufferLength

Type: MQLONG - input

Length of *AuthorityBuffer*. This is the length in bytes of the *AuthorityBuffer* parameter. The authority buffer must be large enough to accommodate the data to be returned.

AuthorityBuffer

Type: MQZAD - output

Authority data. This is the buffer in which the authority data is returned. The buffer must be large enough to accommodate an MQZAD structure, an MQZED structure, plus the longest entity name and longest domain name defined.

Note: Note: This parameter is defined as an MQZAD, as the MQZAD always occurs at the start of the buffer. However, if the buffer is declared as an MQZAD, the buffer will be too small - it must be bigger than an MQZAD so that it can accommodate the MQZAD, MQZED, plus entity and domain names.

AuthorityDataLength

Type: MQLONG - output

Length of data returned in *AuthorityBuffer*. If the authority buffer is too small, *AuthorityDataLength* is set to the length of the buffer required, and the call returns completion code MQCC_FAILED and reason code MQRC_BUFFER_LENGTH_ERROR.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_ENUMERATE_AUTHORITY_DATA, this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_NO_DATA_AVAILABLE

(2379, X'94B') No data available.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_ENUMERATE_AUTHORITY_DATA (QMgrName, StartEnumeration, &Filter,  
                               AuthorityBufferLength,  
                               &AuthorityBuffer,  
                               &AuthorityDataLength, ComponentData,  
                               &Continuation, &CompCode,  
                               &Reason);
```

The parameters passed to the service are declared as follows:

MQCHAR48	QMgrName;	/* Queue manager name */
MQLONG	StartEnumeration;	/* Flag indicating whether call should start enumeration */
MQZAD	Filter;	/* Filter */
MQLONG	AuthorityBufferLength;	/* Length of AuthorityBuffer */
MQZAD	AuthorityBuffer;	/* Authority data */
MQLONG	AuthorityDataLength;	/* Length of data returned in AuthorityBuffer */
MQBYTE	ComponentData[n];	/* Component data */
MQLONG	Continuation;	/* Continuation indicator set by component */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code qualifying CompCode */

MQZ_FREE_USER - Free user

This function is provided by a MQZAS_VERSION_5 authorization service component, and is started by the queue manager to free associated allocated resource.

It is started when an application has finished running under all user contexts, for example during an MQDISC MQI call.

The function identifier for this function (for MQZEP) is MQZID_FREE_USER.

Syntax

MQZ_FREE_USER(QMgrName , FreeParms , ComponentData , Continuation , CompCode , Reason)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

FreeParms

Type: MQZFP - input

Free parameters. A structure containing data relating to the resource to be freed. See [“MQZFP - Free parameters” on page 1684](#) for details.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation flag. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on other components.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_AUTHENTICATE_USER (QMgrName, SecurityParms, ApplicationContext,  
                        IdentityContext, CorrelationPtr, ComponentData,  
                        &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQZFP     FreeParms;          /* Resource to be freed */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_GET_AUTHORITY - Get authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is started by the queue manager to retrieve the authority that an entity has to access the specified object, including (if the entity is a principal) authorities possessed by the groups in which the principal is a member. Authorities from generic profiles are included in the returned authority set.

The function identifier for this function (for MQZEP) is MQZID_GET_AUTHORITY.

Syntax

```
MQZ_GET_AUTHORITY( QMgrName , EntityName , EntityType , ObjectName ,  
ObjectType , Authority , ComponentData , Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityName

Type: MQCHAR12 - input

Entity name. The name of the entity whose access to the object is to be retrieved. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityName*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object to which access is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_GET_AUTHORITY, this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_GET_AUTHORITY (QMgrName, EntityName, EntityType, ObjectName,  
                   ObjectType, &Authority, ComponentData,  
                   &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR12  EntityName;         /* Entity name */  
MQLONG    EntityType;         /* Entity type */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQLONG    Authority;          /* Authority of entity */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_GET_AUTHORITY_2 - Get authority (extended)

This function is provided by a MQZAS_VERSION_2 authorization service component, and is started by the queue manager to retrieve the authority that an entity has to access the specified object.

The function identifier for this function (for MQZEP) is MQZID_GET_AUTHORITY.

MQZ_GET_AUTHORITY_2 is like MQZ_GET_AUTHORITY, but with the *EntityName* parameter replaced by the *EntityData* parameter.

Syntax

```
MQZ_GET_AUTHORITY_2( QMgrName , EntityData , EntityType , ObjectName ,  
                    ObjectType , Authority , ComponentData , Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity for which authorization to the object is to be retrieved. See [“MQZED - Entity descriptor” on page 1681](#) for details.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityData*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object for which the entity authority is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

Syntax

MQZ_GET_AUTHORITY_2 (*QMgrName*, *EntityData*, *EntityType*, *ObjectName*, *ObjectType*, *Authority*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

C invocation

```
MQZ_GET_AUTHORITY_2 (QMgrName, &EntityData, EntityType, ObjectName,
                    ObjectType, &Authority, ComponentData,
                    &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQZED     EntityData;         /* Entity data */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority of entity */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_GET_EXPLICIT_AUTHORITY - Get explicit authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is started by the queue manager to retrieve the authority that an entity has to access the specified object, including (if the entity is a principal) authorities possessed by the groups in which the principal is a member. Authorities from generic profiles are included in the returned authority set.

On UNIX platforms, for the built-in IBM MQ object authority manager (OAM), the returned authority is that possessed only by the principal's primary group.

The function identifier for this function (for MQZEP) is MQZID_GET_EXPLICIT_AUTHORITY.

Syntax

MQZ_GET_EXPLICIT_AUTHORITY(*QMgrName* , *EntityName* , *EntityType* , *ObjectName* , *ObjectType* , *Authority* , *ComponentData* , *Continuation* , *CompCode* , *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityName

Type: MQCHAR12 - input

Entity name. The name of the entity for which access to the object is to be retrieved. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityName*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object for which the entity authority is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_GET_AUTHORITY, this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_GET_EXPLICIT_AUTHORITY (QMgrName, EntityName, EntityType,
                             ObjectName, ObjectType, &Authority,
                             ComponentData, &Continuation,
                             &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR12  EntityName;         /* Entity name */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority of entity */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_GET_EXPLICIT_AUTHORITY_2 - Get explicit authority (extended)

This function is provided by a MQZAS_VERSION_2 authorization service component, and is started by the queue manager to retrieve the authority that a named group has to access a specified object (but

without the additional authority of the **nobody** group), or the authority that the primary group of the named principal has to access a specified object.

The function identifier for this function (for MQZEP) is MQZID_GET_EXPLICIT_AUTHORITY.

MQZ_GET_EXPLICIT_AUTHORITY_2 is like MQZ_GET_EXPLICIT_AUTHORITY, but with the *EntityName* parameter replaced by the *EntityData* parameter.

Syntax

MQZ_GET_EXPLICIT_AUTHORITY_2(*QMgrName* , *EntityData* , *EntityType* , *ObjectName* , *ObjectType* , *Authority* , *ComponentData* , *Continuation* , *CompCode* , *Reason*)

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity whose authorization to the object is to be retrieved. See [“MQZED - Entity descriptor” on page 1681](#) for details.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityData*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object for which the entity authority is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_GET_EXPLICIT_AUTHORITY_2 (QMgrName, &EntityData, EntityType,
                               ObjectName, ObjectType, &Authority,
                               ComponentData, &Continuation,
                               &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQZED     EntityData;         /* Entity data */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority of entity */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_INIT_AUTHORITY - Initialize authorization service

This function is provided by an authorization service component, and is started by the queue manager during configuration of the component. It is expected to call MQZEP in order to provide information to the queue manager.

The function identifier for this function (for MQZEP) is MQZID_INIT_AUTHORITY.

Syntax

```
MQZ_INIT_AUTHORITY( Hconfig , Options , QMgrName , ComponentDataLength ,
                    ComponentData , Version , CompCode , Reason )
```

Parameters***Hconfig***

Type: MQHCONFIG - input

Configuration handle. This handle represents the particular component being initialized. It is to be used by the component when calling the queue manager with the MQZEP function.

Options

Type: MQLONG - input

Initialization options. It must be one of the following values:

MQZIO_PRIMARY

Primary initialization.

MQZIO_SECONDARY

Secondary initialization.

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ComponentDataLength

Type: MQLONG - input

Length of component data. Length in bytes of the *ComponentData* area. This length is defined in the component configuration data.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This is initialized to all zeros before calling the component primary initialization function. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Version

Type: MQLONG - input/output

Version number. On input to the initialization function, this identifies the highest version number that the queue manager supports. The initialization function must change this, if necessary, to the version of the interface which it supports. If on return the queue manager does not support the version returned by the component, it calls the component MQZ_TERM_AUTHORITY function and makes no further use of this component.

The following values are supported:

MQZAS_VERSION_1

Version 1.

MQZAS_VERSION_2

Version 2.

MQZAS_VERSION_3

Version 3.

MQZAS_VERSION_4

Version 4.

MQZAS_VERSION_5

Version 5.

MQZAS_VERSION_6

Version 6.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_INITIALIZATION_FAILED

(2286, X'8EE') Initialization failed for an undefined reason.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_INIT_AUTHORITY (Hconfig, Options, QMgrName, ComponentDataLength,  
                    ComponentData, &Version, &CompCode,  
                    &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG    Hconfig;           /* Configuration handle */  
MQLONG       Options;           /* Initialization options */  
MQCHAR48     QMgrName;         /* Queue manager name */  
MQLONG       ComponentDataLength; /* Length of component data */  
MQBYTE       ComponentData[n]; /* Component data */  
MQLONG       Version;          /* Version number */  
MQLONG       CompCode;         /* Completion code */  
MQLONG       Reason;           /* Reason code qualifying CompCode */
```

MQZ_INQUIRE - Inquire authorization service

This function is provided by a MQZAS_VERSION_5 authorization service component, and is started by the queue manager to query the supported functionality.

Where multiple service components are used, service components are called in reverse order to the order they were installed in.

The function identifier for this function (for MQZEP) is MQZID_INQUIRE.

Syntax

```
MQZ_INQUIRE( QMgrName , SelectorCount , Selectors , IntAttrCount , IntAttrs ,  
             CharAttrLength , CharAttrs , SelectorReturned , ComponentData , Continuation ,  
             CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

SelectorCount

Type: MQLONG - input

Number of selectors. The number of selectors supplied in the *Selectors* parameter.

The value must be in the range 0 through 256.

Selectors

Type: MQLONGxSelectorCount - input

Array of selectors. Each selector identifies a required attribute and must be one of the following:

- MQIACF_INTERFACE_VERSION (integer)
- MQIACF_USER_ID_SUPPORT (integer)
- MQCACF_SERVICE_COMPONENT (character)

Selectors can be specified in any order. The number of selectors in the array is indicated by the *SelectorCount* parameter.

Integer attributes identified by selectors are returned in the *IntAttrs* parameter in the same order as they appear in *Selectors*.

Character attributes identified by selectors are returned in the *CharAttrs* parameter in the same order as they in appear *Selectors*.

IntAttrCount

Type: MQLONG - input

Number of integer attributes supplied in the *IntAttrs* parameter.

The value must be in the range 0 through 256.

IntAttrs

Type: MQLONG x *IntAttrCount* - output

Integer attributes. Array of integer attributes. The integer attributes are returned in the same order as the corresponding integer selectors in the *Selectors* array.

CharAttrCount

Type: MQLONG - input

Length of the character attributes buffer. The length in bytes of the *CharAttrs* parameter.

The value must be at least the sum of the lengths of the requested character attributes. If no character attributes are requested, zero is a valid value.

CharAttrs

Type: MQLONG x *CharAttrCount* - output

Character attributes buffer. Buffer containing character attributes, concatenated together. The character attributes are returned in the same order as the corresponding character selectors in the *Selectors* array.

The length of the buffer is given by the *CharAttrCount* parameter.

SelectorReturned

Type: MQLONG x *SelectorCount* - input

Selector returned. Array of values identifying which attributes have been returned from the set requested for by the selectors in the *Selectors* parameter. The number of values in this array is indicated by the *SelectorCount* parameter. Each value in the array relates to the selector from the corresponding position in the *Selectors* array. Each value is one of the following:

MQZSL_RETURNED

The attribute requested by the corresponding selector in the *Selectors* parameter has been returned.

MQZSL_NOT_RETURNED

The attribute requested by the corresponding selector in the *Selectors* parameter has not been returned.

The array is initialized with all values as *MQZSL_NOT_RETURNED*. When an authorization service component returns an attribute, it sets the appropriate value in the array to *MQZSL_NOT_RETURNED*. This allows any other authorization service components, to which the inquire call is made, to identify which attributes have already been returned.

ComponentData

Type: MQBYTE x *ComponentDataLength* - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CHAR_ATTRS_TOO_SHORT

Not enough space for character attributes.

MQRC_INT_COUNT_TOO_SMALL

Not enough space for integer attributes.

If *CompCode* is MQCC_FAILED:

MQRC_SELECTOR_COUNT_ERROR

Number of selectors is not valid.

MQRC_SELECTOR_ERROR

Attribute selector not valid.

MQRC_SELECTOR_LIMIT_EXCEEDED

Too many selectors specified.

MQRC_INT_ATTR_COUNT_ERROR

Number of integer attributes is not valid.

MQRC_INT_ATTRS_ARRAY_ERROR

Integer attributes array not valid.

MQRC_CHAR_ATTR_LENGTH_ERROR

Number of character attributes is not valid.

MQRC_CHAR_ATTRS_ERROR

Character attributes string is not valid.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_INQUIRE (QMgrName, SelectorCount, Selectors, IntAttrCount,
              &IntAttrs, CharAttrLength, &CharAttrs,
              SelectorReturned, ComponentData, &Continuation,
              &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48    QMgrName;           /* Queue manager name */
MQLONG       SelectorCount;      /* Selector count */
MQLONG       Selectors[n];       /* Selectors */
MQLONG       IntAttrCount;       /* IntAttrs count */
MQLONG       IntAttrs[n];        /* Integer attributes */
MQLONG       CharAttrCount;      /* CharAttrs count */
MQLONG       CharAttrs[n];       /* Character attributes */
MQLONG       SelectorReturned[n]; /* Selector returned */
MQBYTE       ComponentData[n];   /* Component data */
MQLONG       Continuation;       /* Continuation indicator set by
                                component */
MQLONG       CompCode;           /* Completion code */
MQLONG       Reason;            /* Reason code qualifying CompCode */
```

MQZ_REFRESH_CACHE - Refresh all authorizations

This function is provided by an MQZAS_VERSION_3 authorization service component, and is invoked by the queue manager to refresh the list of authorizations held internally by the component.

The function identifier for this function (for MQZEP) is MQZID_REFRESH_CACHE (8L).

Syntax

`MQZ_REFRESH_CACHE(QMgrName , ComponentData , Continuation , CompCode , Reason)`

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

C invocation

```
MQZ_REFRESH_CACHE (QMgrName, ComponentData,
                  &Continuation, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG     Continuation;       /* Continuation indicator set by
                               component */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQZ_SET_AUTHORITY - Set authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is started by the queue manager to set the authority that an entity has to access the specified object.

The function identifier for this function (for MQZEP) is MQZID_SET_AUTHORITY.

Note: This function overrides any existing authorities. To preserve any existing authorities you must set them again with this function.

Syntax

```
MQZ_SET_AUTHORITY( QMgrName , EntityName , EntityType , ObjectName ,
                  ObjectType , Authority , ComponentData , Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityName

Type: MQCHAR12 - input

Entity name. The name of the entity for which access to the object is to be retrieved. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityName*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object to which access is required. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If one authority is being set, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authority is being set, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_GET_AUTHORITY, this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_SET_AUTHORITY (QMgrName, EntityName, EntityType, ObjectName,  
                  ObjectType, Authority, ComponentData,  
                  &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR12  EntityName;         /* Entity name */  
MQLONG    EntityType;         /* Entity type */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQLONG    Authority;          /* Authority to be checked */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_SET_AUTHORITY_2 - Set authority (extended)

This function is provided by a MQZAS_VERSION_2 authorization service component, and is started by the queue manager to set the authority that an entity has to access the specified object.

The function identifier for this function (for MQZEP) is MQZID_SET_AUTHORITY.

Note: This function overrides any existing authorities. To preserve any existing authorities you must set them again with this function.

MQZ_SET_AUTHORITY_2 is like MQZ_SET_AUTHORITY, but with the *EntityName* parameter replaced by the *EntityData* parameter.

Syntax

```
MQZ_SET_AUTHORITY_2( QMgrName , EntityData , EntityType , ObjectName ,  
                   ObjectType , Authority , ComponentData , Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity whose authorization to the object is to be set. See [“MQZED - Entity descriptor”](#) on page 1681 for details.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by *EntityData*. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName

Type: MQCHAR48 - input

Object name. The name of the object to which the entity authority is to be set. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType

Type: MQLONG - input

Object type. The type of entity specified by *ObjectName*. It must be one of the following values:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

MQOT_TOPIC

Topic.

Authority

Type: MQLONG - input

Authority of entity. If one authority is being set, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authority is being set, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_SET_AUTHORITY_2 (QMgrName, &EntityData, EntityType, ObjectName,
                    ObjectType, Authority, ComponentData,
                    &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQZED     EntityData;         /* Entity data */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority to be checked */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_TERM_AUTHORITY - Terminate authorization service

This function is provided by an authorization service component, and is started by the queue manager when it no longer requires the services of this component. The function must perform any cleanup required by the component.

The function identifier for this function (for MQZEP) is MQZID_TERM_AUTHORITY.

Syntax

MQZ_TERM_AUTHORITY(Hconfig , Options , QMgrName , ComponentData , CompCode , Reason)

Parameters

Hconfig

Type: MQHCONFIG - input

Configuration handle. This handle represents the particular component being terminated. It is to be used by the component when calling the queue manager with the MQZEP function.

Options

Type: MQLONG - input

Termination options. It must be one of the following values:

MQZTO_PRIMARY

Primary termination.

MQZTO_SECONDARY

Secondary termination.

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter on the MQZ_INIT_AUTHORITY call.

When the MQZ_TERM_AUTHORITY call has completed, the queue manager discards this data.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_TERMINATION_FAILED

(2287, X'8FF') Termination failed for an undefined reason.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_TERM_AUTHORITY (Hconfig, Options, QMgrName, ComponentData,  
                    &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG  Hconfig;           /* Configuration handle */  
MQLONG     Options;          /* Termination options */  
MQCHAR48   QMgrName;        /* Queue manager name */  
MQBYTE     ComponentData[n]; /* Component data */  
MQLONG     CompCode;         /* Completion code */  
MQLONG     Reason;          /* Reason code qualifying CompCode */
```

MQZ_DELETE_NAME - Delete name

This function is provided by a name service component, and is started by the queue manager to delete an entry for the specified queue.

The function identifier for this function (for MQZEP) is MQZID_DELETE_NAME.

Syntax

```
MQZ_DELETE_NAME( QMgrName , QName , ComponentData , Continuation , CompCode ,  
Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

QName

Type: MQCHAR48 - input

Queue name. The name of the queue for which an entry is to be deleted. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the ComponentDataLength parameter on the MQZ_INIT_NAME call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. It must be one of the following values:

MQZCI_DEFAULT

Continuation dependent on queue manager.

MQZCI_STOP

Do not continue with next component.

For the **MQZ_DELETE_NAME** command, the queue manager does not attempt to start another component, no matter what is returned in the **Continuation** parameter.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_WARNING

Warning (partial completion).

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_UNKNOWN_NAME

(2288, X'8F0') Queue name not found.

Note: It might not be possible to return this code if the underlying service responds with success for this case.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_DELETE_NAME (QMgrName, QName, ComponentData, &Continuation,  
                 &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

MQCHAR48	QMgrName;	/* Queue manager name */
MQCHAR48	QName;	/* Queue name */
MQBYTE	ComponentData[n];	/* Component data */
MQLONG	Continuation;	/* Continuation indicator set by component */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code qualifying CompCode */

MQZ_INIT_NAME - Initialize name service

This function is provided by a name service component, and is started by the queue manager during configuration of the component. It is expected to call MQZEP in order to provide information to the queue manager.

The function identifier for this function (for MQZEP) is MQZID_INIT_NAME.

Syntax

MQZ_INIT_NAME(*Hconfig* , *Options* , *QMgrName* , *ComponentDataLength* ,
ComponentData , *Version* , *CompCode* , *Reason*)

Parameters

Hconfig

Type: MQHCONFIG - input

Configuration handle. This handle represents the particular component being initialized. It is to be used by the component when calling the queue manager with the MQZEP function.

Options

Type: MQLONG - input

Initialization options. It must be one of the following values:

MQZIO_PRIMARY

Primary initialization.

MQZIO_SECONDARY

Secondary initialization.

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ComponentDataLength

Type: MQLONG - input

Length of component data. Length in bytes of the *ComponentData* area. This length is defined in the component configuration data.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This is initialized to all zeros before calling the component primary initialization function. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Version

Type: MQLONG - input/output

Version number. On input to the initialization function, this identifies the highest version number that the queue manager supports. The initialization function must change this, if necessary, to the version of the interface which it supports. If on return the queue manager does not support the version returned by the component, it calls the component MQZ_TERM_NAME function and makes no further use of this component.

The following values are supported:

MQZAS_VERSION_1

Version 1.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_INITIALIZATION_FAILED

(2286, X'8EE') Initialization failed for an undefined reason.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_INIT_NAME (Hconfig, Options, QMgrName, ComponentDataLength,
               ComponentData, &Version, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG  Hconfig;           /* Configuration handle */
MQLONG     Options;           /* Initialization options */
MQCHAR48   QMgrName;         /* Queue manager name */
MQLONG     ComponentDataLength; /* Length of component data */
MQBYTE     ComponentData[n];  /* Component data */
MQLONG     Version;           /* Version number */
MQLONG     CompCode;          /* Completion code */
MQLONG     Reason;            /* Reason code qualifying CompCode */
```

MQZ_INSERT_NAME - Insert name

This function is provided by a name service component, and is started by the queue manager to insert an entry for the specified queue, containing the name of the queue manager that owns the queue. If the queue is already defined in the service, the call fails.

The function identifier for this function (for MQZEP) is MQZID_INSERT_NAME.

Syntax

```
MQZ_INSERT_NAME( QMgrName , QName , ResolvedQMgrName , ComponentData ,
                 Continuation , CompCode , Reason )
```

Parameters**QMgrName**

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

QName

Type: MQCHAR48 - input

Queue name. The name of the queue for which an entry is to be inserted. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

ResolvedQMgrName

Type: MQCHAR48 - input

Resolved queue manager name. The name of the queue manager to which the queue resolves. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_NAME call.

Continuation

Type: MQLONG - input/output

Continuation indicator set by component. For MQZ_INSERT_NAME, the queue manager does not attempt to start another component, whatever is returned in the *Continuation* parameter.

The following values are supported:

MQZCI_DEFAULT

Continuation dependent on queue manager.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_Q_ALREADY_EXISTS

(2290, X'8F2') Queue object already exists.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_INSERT_NAME (QMgrName, QName, ResolvedQMgrName, ComponentData,  
                &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR48  QName;             /* Queue name */  
MQCHAR48  ResolvedQMgrName;  /* Resolved queue manager name */  
MQBYTE    ComponentData[n];  /* Component data */  
MQLONG    Continuation;      /* Continuation indicator set by  
                             component */  
MQLONG    CompCode;          /* Completion code */  
MQLONG    Reason;            /* Reason code qualifying CompCode */
```

MQZ_LOOKUP_NAME - Lookup name

This function is provided by a name service component, and is started by the queue manager to retrieve the name of the owning queue manager, for a specified queue.

The function identifier for this function (for MQZEP) is MQZID_LOOKUP_NAME.

Syntax

```
MQZ_LOOKUP_NAME( QMgrName , QName , ResolvedQMgrName , ComponentData ,  
Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

QName

Type: MQCHAR48 - input

Queue name. The name of the queue for which an entry is to be resolved. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

ResolvedQMgrName

Type: MQCHAR48 - output

Resolved queue manager name. If the function completes successfully, this is the name of the queue manager that owns the queue.

The name returned by the service component must be padded on the right with blanks to the full length of the parameter; the name must not be terminated by a null character, or contain leading or embedded blanks.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_NAME call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. For MQZ_LOOKUP_NAME, the queue manager specifies whether to start another name service component, as follows:

- If *CompCode* is MQCC_OK, no further components are started, whatever value is returned in *Continuation*.
- If *CompCode* is not MQCC_OK, a further component is started, unless *Continuation* is MQZCI_STOP.

The following values are supported:

MQZCI_DEFAULT

Continuation dependent on queue manager.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_Q_NAME

(2288, X'8F0') Queue name not found.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_LOOKUP_NAME (QMgrName, QName, ResolvedQMgrName, ComponentData,
                  &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR48  QName;              /* Queue name */
MQCHAR48  ResolvedQMgrName;    /* Resolved queue manager name */
MQBYTE    ComponentData[n];    /* Component data */
MQLONG    Continuation;        /* Continuation indicator set by
                                component */
MQLONG    CompCode;            /* Completion code */
MQLONG    Reason;              /* Reason code qualifying CompCode */
```

MQZ_TERM_NAME - Terminate name service

This function is provided by a name service component, and is started by the queue manager when it no longer requires the services of this component. The function must perform any cleanup required by the component.

The function identifier for this function (for MQZEP) is MQZID_TERM_NAME.

Syntax

MQZ_TERM_NAME(*Hconfig* , *Options* , *QMgrName* , *ComponentData* , *CompCode* , *Reason*)

Parameters

Hconfig

Type: MQHCONFIG - input

Configuration handle. This handle represents the particular component being terminated. It is used by the component when calling the queue manager with the MQZEP function.

Options

Type: MQLONG - input

Termination options. It must be one of the following values:

MQZTO_PRIMARY

Primary termination.

MQZTO_SECONDARY

Secondary termination.

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ComponentData

Type: MQBYTE x ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of these component functions is called.

Component data is in shared memory accessible to all processes.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_NAME call.

When the MQZ_TERM_NAME call has completed, the queue manager discards this data.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_TERMINATION_FAILED

(2287, X'8FF') Termination failed for an undefined reason.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_TERM_NAME (Hconfig, Options, QMgrName, ComponentData, &CompCode,  
               &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG  Hconfig;           /* Configuration handle */  
MQLONG     Options;           /* Termination options */  
MQCHAR48   QMgrName;          /* Queue manager name */  
MQBYTE     ComponentData[n]; /* Component data */  
MQLONG     CompCode;          /* Completion code */  
MQLONG     Reason;            /* Reason code qualifying CompCode */
```

MQZAC - Application context

The MQZAC structure is used on the MQZ_AUTHENTICATE_USER call for the *ApplicationContext* parameter. This parameter specifies data related to the calling application.

[Table 1](#) summarizes the fields in the structure.

Table 658. Fields in MQZAC	
Field	Description
StrucId	Structure identifier
Version	Structure version number
ProcessId	Process identifier
ThreadId	Thread identifier
ApplName	Application name
UserID	User identifier
EffectiveUserID	Effective user identifier
Environment	Environment
CallerType	Caller type
AuthenticationType	Authentication type
BindType	Bind type

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQZAC_STRUC_ID

Identifier for application context structure.

For the C programming language, the constant MQZAC_STRUC_ID_ARRAY is also defined; this has the same value as MQZAC_STRUC_ID, but is an array of characters instead of a string.

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQZAC_VERSION_1

Version-1 application context structure. The constant MQZAC_CURRENT_VERSION specifies the version number of the current version.

ProcessId

Type: MQPID - input

Process identifier of the application.

ThreadId

Type: MQTID - input

Thread identifier of the application.

ApplName

Type: MQCHAR28 - input

Application name.

UserID

Type: MQCHAR12 - input

User identifier. On UNIX systems this field specifies the application's real user ID. On Windows this field specifies the application's user ID.

EffectiveUserID

Type: MQCHAR12 - input

Effective user identifier. On UNIX systems this field specifies the application's effective user ID. On Windows this field is blank.

Environment

Type: MQLONG - input

Environment. This field specifies the environment from which the call was made. The field is one of the following values:

MQXE_COMMAND_SERVER

Command server

MQXE_MQSC

runmqsc command interpreter

MQXE_MCA

Message channel agent MQXE_OTHER

MQXE_OTHER

Undefined environment

CallerType

Type: MQLONG - input

Caller Type. This field specifies the type of program that made the call. The field is one of the following values:

MQXACT_EXTERNAL

The call is external to the queue manager.

MQXACT_INTERNAL

The call is internal to the queue manager.

AuthenticationType

Type: MQLONG - input

Authentication Type. This field specifies the type of authentication being performed. The field is one of the following values:

MQZAT_INITIAL_CONTEXT

The authentication call is due to user context being initialized. This value is used during an MQCONN or MQCONNEX call.

MQZAT_CHANGE_CONTEXT

The authentication call is due to the user context being changed. This value is used when the MCA changes the user context. Parent topic: MQZAC -

BindType

Type: MQLONG - input

Bind Type. This field specifies the type of binding in use. The field is one of the following values:

MQCNO_FASTPATH_BINDING

Fastpath binding.

MQCNO_SHARED_BINDING

Shared binding.

MQCNO_ISOLATED_BINDING

Isolated binding.

C declaration

Declare the fields of the structure as follows:

```
typedef struct tagMQZAC MQZAC;
struct tagMQZAC {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQPID      ProcessId;        /* Process identifier */
    MQTID      ThreadId;         /* Thread identifier */
    MQCHAR28   ApplName;         /* Application name */
    MQCHAR12   UserID;           /* User identifier */
    MQCHAR12   EffectiveUserID;   /* Effective user identifier */
    MQLONG     Environment;       /* Environment */
    MQLONG     CallerType;        /* Caller type */
    MQLONG     AuthenticationType; /* Authentication type */
    MQLONG     BindType;          /* Bind type */
};
```

MQZAD - Authority data

The MQZAD structure is used on the MQZ_ENUMERATE_AUTHORITY_DATA call for two parameters, one input and one output.

- MQZAD is used for the **Filter** parameter which is input to the call. This parameter specifies the selection criteria that are to be used to select the authority data returned by the call.
- MQZAD is also used for the **AuthorityBuffer** parameter which is output from the call. This parameter specifies the authorizations for one combination of profile name, object type, and entity.

Table 1. summarizes the fields in the structure.

Table 659. Fields in MQZAD	
Field	Description
<u>StrucId</u>	Structure identifier
<u>Version</u>	Structure version number

Table 659. Fields in MQZAD (continued)	
Field	Description
<u>ProfileName</u>	Profile name
<u>ObjectType</u>	Object type
<u>Authority</u>	Authority
<u>EntityDataPtr</u>	Pointer to entity data
<u>EntityType</u>	Entity type
<u>Options</u>	Options

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQZAD_STRUC_ID

Identifier for authority data structure.

For the C programming language, the constant MQZAD_STRUC_ID_ARRAY is also defined; this has the same value as MQZAD_STRUC_ID, but is an array of characters instead of a string.

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQZAD_VERSION_1

Version-1 application context structure. The constant MQZAD_CURRENT_VERSION specifies the version number of the current version.

The following constant specifies the version number of the current version:

MQZAD_CURRENT_VERSION

Current version of authority data structure.

ProfileName

Type: MQCHAR48 - input

Profile name.

For the **Filter** parameter, this field is the profile name for which authority data is required. If the name is entirely blank up to the end of the field or the first null character, authority data for all profile names is returned.

For the **AuthorityBuffer** parameter, this field is the name of a profile that matches the specified selection criteria.

ObjectType

Type: MQLONG - input

Object type.

For the **Filter** parameter, this field is the object type for which authority data is required. If the value is MQOT_ALL, authority data for all object types is returned.

For the **AuthorityBuffer** parameter, this field is the object type to which the profile identified by the **ProfileName** parameter applies.

The value is one of the following; for the **Filter** parameter, the value MQOT_ALL is also valid:

MQOT_AUTH_INFO

Authentication information

MQOT_CHANNEL

Channel

MQOT_CLNTCONN_CHANNEL

Client connection channel

MQOT_LISTENER

Listener

MQOT_NAMELIST

Namelist

MQOT_PROCESS

Process definition

MQOT_Q

Queue

MQOT_Q_MGR

Queue manager

MQOT_SERVICE

Service

Authority

Type: MQLONG - input

Authority.

For the **Filter** parameter, this field is ignored.

For the **AuthorityBuffer** parameter, this field represents the authorizations that the entity has to the objects identified by **ProfileName** and **ObjectType**. If the entity has only one authority, the field is equal to the appropriate authorization value (MQZAO_* constant). If the entity has more than one authority, the field is the bitwise OR of the corresponding MQZAO_* constants.

EntityDataPtr

Type: PMQZED - input

Address of MQZED structure identifying an entity.

For the **Filter** parameter, this field points to an MQZED structure that identifies the entity for which authority data is required. If **EntityDataPtr** is the null pointer, authority data for all entities is returned.

For the **AuthorityBuffer** parameter, this field points to an MQZED structure that identifies the entity for which authority data has been returned.

EntityType

Type: MQLONG - input

Entity type.

For the **Filter** parameter, this field specifies the entity type for which authority data is required. If the value is MQZAET_NONE, authority data for all entity types is returned.

For the **AuthorityBuffer** parameter, this field specifies the type of the entity identified by the MQZED structure pointed to by the **EntityDataPtr** parameter.

The value is one of the following; for the **Filter** parameter, the value MQZAET_NONE is also valid:

MQZAET_PRINCIPAL

Principal

MQZAET_GROUP

Group

Options

Type: MQAUTHOPT - input

Options. This field specifies options that give control over the profiles that are displayed. One of the following values must be specified:

MQAUTHOPT_NAME_ALL_MATCHING

Displays all profiles

MQAUTHOPT_NAME_EXPLICIT

Displays profiles that have exactly the same name as specified in the **ProfileName** field.

In addition, one of the following must also be specified:

MQAUTHOPT_ENTITY_SET

Display all profiles that are used to calculate the cumulative authority that the entity has to the object specified by the **ProfileName** parameter. The **ProfileName** parameter must not contain any wildcard characters.

- If the specified entity is a principal, for each member of the set {entity, groups} the most applicable profile that applies to the object is displayed.
- If the specified entity is a group, the most applicable profile from the group that applies to the object is displayed.
- If this value is specified, then the values of **ProfileName**, **ObjectType**, **EntityType**, and the entity name specified in the **EntityDataPtr** MQZED structure, must all be non-blank.

If you have specified MQAUTHOPT_NAME_ALL_MATCHING, you can also specify the following value:

MQAUTHOPT_ENTITY_EXPLICIT

Displays profiles that have exactly the same entity name as the entity name specified in the **EntityDataPtr** MQZED structure.

C declaration

```
typedef struct tagMQZAD MQZAD;  
struct tagMQZAD {  
    MQCHAR4    StrucId;        /* Structure identifier */  
    MQLONG     Version;        /* Structure version number */  
    MQCHAR48    ProfileName;    /* Profile name */  
    MQLONG     ObjectType;      /* Object type */  
    MQLONG     Authority;       /* Authority */  
    PMQZED     EntityDataPtr;   /* Address of MQZED structure identifying an  
                                entity */  
    MQLONG     EntityType;      /* Entity type */  
    MQAUTHOPT   Options;        /* Options */  
};
```

MQZED - Entity descriptor

The MQZED structure is used in a number of authorization service calls to specify the entity for which authorization is to be checked.

Table 1. summarizes the fields in the structure.

Table 660. Fields in MQZED	
Field	Description
StrucId	Structure identifier
Version	Version
EntityName Ptr	Entity name
EntityDomainPtr	Entity domain pointer
SecurityId	Security identifier

Table 660. Fields in MQZED (continued)	
Field	Description
<u>CorrelationPtr</u>	Correlation pointer

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQZED_STRUC_ID

Identifier for entity descriptor structure.

For the C programming language, the constant MQZED_STRUC_ID_ARRAY is also defined; this has the same value as MQZED_STRUC_ID, but is an array of characters instead of a string.

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQZED_VERSION_1

Version-1 entity descriptor structure.

The following constant specifies the version number of the current version:

MQZED_CURRENT_VERSION

Current version of entity descriptor structure.

EntityNamePtr

Type: PMQCHAR - input

Profile name.

Address of entity name. This is a pointer to the name of the entity whose authorization is to be checked.

EntityDomainPtr

Type: PMQCHAR - input

Address of entity domain name. This is a pointer to the name of the domain containing the definition of the entity whose authorization is to be checked.

SecurityId

Type: MQBYTE40 - input

Authority.

Security identifier. This is the security identifier whose authorization is to be checked.

CorrelationPtr

Type: MQPTR - input

Correlation pointer. This facilitates the passing of correlational data between the authenticate user function and other appropriate OAM functions.

C declaration

```
typedef struct tagMQZED MQZED;
struct tagMQZED {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    PMQCHAR    EntityNamePtr;    /* Address of entity name */
    PMQCHAR    EntityDomainPtr;  /* Address of entity domain name */
    MQBYTE40   SecurityId;       /* Security identifier */
    MQPTR      CorrelationPtr;   /* Address of correlation data */
}
```

Fields

MQZEP - Add component entry point

A service component starts this function, during initialization, to add an entry point to the entry point vector for that service component.

Syntax

MQZEP (*Hconfig*, *Function*, *EntryPoint*, *CompCode*, *Reason*)

Parameters

Hconfig

Type: MQHCONFIG - input

Configuration handle. This handle represents the component that is being configured for this particular installable service. It must be the same as the component passed to the component configuration function by the queue manager on the component initialization call.

Function

Type: MQLONG - input

Function identifier. Valid values for this are defined for each installable service.

If MQZEP is called more than once for the same function, the last call made provides the entry point that is used.

EntryPoint

Type: PMQFUNC - input

Function entry point. This is the address of the entry point provided by the component to perform the function.

The value NULL is valid, and indicates that the function is not provided by this component. NULL is assumed for entry points that are not defined using MQZEP.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_FUNCTION_ERROR

(2281, X'8E9') Function identifier not valid.

MQRC_HCONFIG_ERROR

(2280, X'8E8') Configuration handle not valid.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZEP (Hconfig, Function, EntryPoint, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONFIG  Hconfig;      /* Configuration handle */
MQLONG     Function;     /* Function identifier */
PMQFUNC    EntryPoint;   /* Function entry point */
MQLONG     CompCode;     /* Completion code */
MQLONG     Reason;       /* Reason code qualifying CompCode */
```

MQZFP - Free parameters

The MQZFP structure is used on the MQZ_FREE_USER call for the *FreeParms* parameter. This parameter specifies data related to resource to be freed.

Table 1. summarizes the fields in the structure.

Table 661. Fields in MQZFP	
Field	Description
<u>StrucId</u>	Structure identifier
<u>Version</u>	Version
<u>Reserved</u>	Reserved field
<u>CorrelationPtr</u>	Correlation pointer

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQZIC_STRUC_ID

Identifier for identity context structure. For the C programming language, the constant MQZIC_STRUC_ID_ARRAY is also defined; this has the same value as MQZIC_STRUC_ID, but is an array of characters instead of a string.

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQZFP_VERSION_1

Version-1 free parameters structure.

The following constant specifies the version number of the current version:

MQZFP_CURRENT_VERSION

Current version of free parameters structure.

Reserved

Type: MQBYTE8 - input

Reserved field. The initial value is null.

CorrelationPtr

Type: MQPTR - input

Correlation pointer. Address of correlation data relating to the resource to be freed.

C declaration

```
typedef struct tagMQZFP MQZFP;
struct tagMQZFP {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;          /* Structure version number */
    MQBYTE8    Reserved;         /* Reserved field */
    MQPTR      CorrelationPtr;   /* Address of correlation data */
};
```

Fields

MQZIC - Identity context

The MQZIC structure is used on the MQZ_AUTHENTICATE_USER call for the *IdentityContext* parameter.

The MQZIC structure contains identity context information, which identifies the user of the application that first put the message on a queue:

- The queue manager fills the *UserIdentifier* field with a name that identifies the user, the way that the queue manager can do this depends on the environment in which the application is running.
- The queue manager fills the *AccountingToken* field with a token or number that it determined from the application that put the message.
- Applications can use the *ApplIdentityData* field for any extra information that they want to include about the user (for example, an encrypted password).

Suitably authorized applications can set the identity context using the MQZ_AUTHENTICATE_USER function.

A Windows systems security identifier (SID) is stored in the *AccountingToken* field when a message is created under IBM MQ for Windows. The SID can be used to supplement the *UserIdentifier* field and to establish the credentials of a user.

Table 1. summarizes the fields in the structure.

Table 662. Fields in MQZIC	
Field	Description
StrucId	Structure identifier
Version	Version
UserIdentifier	User identifier
AccountingToken	Accounting token
ApplIdentityData	Application identity data

Fields

StrucId

Type: MQCHAR4 - input

Structure identifier. The value is as follows:

MQZIC_STRUC_ID

Identifier for identity context structure. For the C programming language, the constant MQZIC_STRUC_ID_ARRAY is also defined; this has the same value as MQZIC_STRUC_ID, but is an array of characters instead of a string.

Version

Type: MQLONG - input

Structure version number. The value is as follows:

MQZIC_VERSION_1

Version-1 identity context structure.

The following constant specifies the version number of the current version:

MQZIC_CURRENT_VERSION

Current version of identity context structure.

UserIdentifier

Type: MQCHAR12 - input

User identifier. This is part of the identity context of the message. *UserIdentifier* specifies the user identifier of the application that originated the message. The queue manager treats this information as character data, but does not define the format of it. For more information on the *UserIdentifier* field, see [“UserIdentifier \(MQCHAR12\)” on page 441](#).

AccountingToken

Type: MQBYTE32 - input

Accounting token. This is part of the identity context of the message. *AccountingToken* allows an application to cause work done as a result of the message to be appropriately charged. The queue manager treats this information as a string of bits and does not check its content. For more information on the *AccountingToken* field, see [“AccountingToken \(MQBYTE32\)” on page 400](#).

ApplIdentityData

Type: MQCHAR32 - input

Application data relating to identity. This is part of the identity context of the message. *ApplIdentityData* is information that is defined by the application suite that can be used to provide additional information about the origin of the message. For example, it could be set by applications running with suitable user authority to indicate whether the identity data is trusted. For more information on the *ApplIdentityData* field, see [“ApplIdentityData \(MQCHAR32\)” on page 401](#).

C declaration

```
typedef struct tagMQZED MQZED;
struct tagMQZED {
    MQCHAR4    StrucId;          /* Structure identifier */
    MQLONG     Version;         /* Structure version number */
    MQCHAR12   UserIdentifier;   /* User identifier */
    MQBYTE32   AccountingToken; /* Accounting token */
    MQCHAR32   ApplIdentityData; /* Application data relating to identity */
};
```

Fields

Installable services interface reference information

Use this information to understand the reference information for the installable services.

For each function there is a description, including the function identifier (for MQZEP).

The *parameters* are shown listed in the order they must occur. They must all be present.

Each parameter name is followed by its data type in parentheses. These are the elementary data types described in [“Elementary data types” on page 959](#).

The C language invocation is also given, after the description of the parameters.

MQZEP - Add component entry point

This function is invoked by a service component, during initialization, to add an entry point to the entry point vector for that service component.

Syntax

```
MQZEP (Hconfig, Function, EntryPoint, CompCode, Reason)
```

Parameters

The MQZEP call has the following parameters.

Hconfig (MQHCONFIG) - input

Configuration handle.

This handle represents the component which is being configured for this particular installable service. It must be the same as the one passed to the component configuration function by the queue manager on the component initialization call.

Function (MQLONG) - input

Function identifier.

Valid values for this are defined for each installable service. If MQZEP is called more than once for the same function, the last call made provides the entry point which is used.

EntryPoint (PMQFUNC) - input

Function entry point.

This is the address of the entry point provided by the component to perform the function. The value NULL is valid, and indicates that the function is not provided by this component. NULL is assumed for entry points which are not defined using MQZEP.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_FUNCTION_ERROR

(2281, X'8E9') Function identifier not valid.

MQRC_HCONFIG_ERROR

(2280, X'8E8') Configuration handle not valid.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZEP (Hconfig, Function, EntryPoint, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONFIG  Hconfig;      /* Configuration handle */
MQLONG     Function;     /* Function identifier */
PMQFUNC    EntryPoint;   /* Function entry point */
MQLONG     CompCode;     /* Completion code */
MQLONG     Reason;       /* Reason code qualifying CompCode */
```

MQHCONFIG - Configuration handle

The MQHCONFIG data type represents a configuration handle, that is, the component that is being configured for a particular installable service. A configuration handle must be aligned on its natural boundary.

Applications must test variables of this type for equality only.

C declaration

```
typedef void MQPOINTER MQHCONFIG;
```

PMQFUNC - Pointer to function

Pointer to a function.

C declaration

```
typedef void MQPOINTER PMQFUNC;
```

MQZ_AUTHENTICATE_USER - Authenticate user

This function is provided by a MQZAS_VERSION_5 authorization service component. It is invoked by the queue manager to authenticate a user, or to set identity context fields.

It is invoked when an IBM MQ user application context is established. This happens during connect calls at the point where the application's user context is initialized, and at each point where the application's user context is changed. Each time a connect call is made, the application's user context information is reacquired in the *IdentityContext* field.

The function identifier for this function (for MQZEP) is MQZID_AUTHENTICATE_USER.

Syntax

MQZ_AUTHENTICATE_USER (*QMgrName*, *SecurityParms*, *ApplicationContext*, *IdentityContext*, *CorrelationPtr*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_AUTHENTICATE_USER call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character. The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

SecurityParms (MQCSP) - input

Security parameters.

Data relating to the user ID, password, and authentication type.

During an MQCONN MQI call this parameter contains null, or default values.

ApplicationContext (MQZAC) - input

Application context.

Data relating to the calling application. See [“MQZAC - Application context”](#) on page 1717 for details. During every MQCONN or MQCONNX MQI call, the user context information in the MQZAC structure is reacquired.

IdentityContext (MQZIC) - input/output

Identity context.

On input to the authenticate user function, this identifies the current identity context. The authenticate user function can change this, at which point the queue manager adopts the new identity context. See [“MQZIC - Identity context”](#) on page 1724 for more details on the MQZIC structure.

CorrelationPtr (MQPTR) - output

Correlation pointer.

Specifies the address of any correlation data. This pointer is then passed on to other OAM calls.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this components functions is called. The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation flag.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on other components.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information about these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_AUTHENTICATE_USER (QMgrName, SecurityParms, ApplicationContext,  
                        IdentityContext, &CorrelationPtr, ComponentData,  
                        &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```

MQCHAR48  QMgrName;           /* Queue manager name */
MQCSP     SecurityParms;      /* Security parameters */
MQZAC     ApplicationContext; /* Application context */
MQZIC     IdentityContext;    /* Identity context */
MQPTR     CorrelationPtr;     /* Correlation pointer */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */

```

MQZ_CHECK_AUTHORITY - Check authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is invoked by the queue manager to check whether an entity has authority to perform a particular action, or actions, on a specified object.

The function identifier for this function (for MQZEP) is MQZID_CHECK_AUTHORITY.

Syntax

MQZ_CHECK_AUTHORITY (*QMgrName*, *EntityName*, *EntityType*,
ObjectName, *ObjectType*, *Authority*, *ComponentData*, *Continuation*, *CompCode*,
Reason)

Parameters

The MQZ_CHECK_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character. The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

EntityName (MQCHAR12) - input

Entity name.

The name of the entity whose authorization to the object is to be checked. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

It is not essential for this entity to be known to the underlying security service. If it is not known, then the authorizations of the special **nobody** group (to which all entities are assumed to belong) are used for the check. An all-blank name is valid and can be used in this way.

EntityType (MQLONG) - input

Entity type.

The type of entity specified by *EntityName*. It is one of the following:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName (MQCHAR48) - input

Object name.

The name of the object to which access is required. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType (MQLONG) - input

Object type.

The type of entity specified by *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority (MQLONG) - input

Authority to be checked.

If one authorization is being checked, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authorization is being checked, it is the bitwise OR of the corresponding MQZAO_* constants.

The following authorizations apply to use of the MQI calls:

MQZAO_CONNECT

Ability to use the MQCONN call.

MQZAO_BROWSE

Ability to use the MQGET call with a browse option.

This allows the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_MSG_UNDER_CURSOR, or MQGMO_BROWSE_NEXT option to be specified on the MQGET call.

MQZAO_INPUT

Ability to use the MQGET call with an input option.

This allows the MQOO_INPUT_SHARED, MQOO_INPUT_EXCLUSIVE, or MQOO_INPUT_AS_Q_DEF option to be specified on the MQOPEN call.

MQZAO_OUTPUT

Ability to use the MQPUT call.

This allows the MQOO_OUTPUT option to be specified on the MQOPEN call.

MQZAO_INQUIRE

Ability to use the MQINQ call.

This allows the MQOO_INQUIRE option to be specified on the MQOPEN call.

MQZAO_SET

Ability to use the MQSET call.

This allows the MQOO_SET option to be specified on the MQOPEN call.

MQZAO_PASS_IDENTITY_CONTEXT

Ability to pass identity context.

This allows the MQOO_PASS_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_PASS_ALL_CONTEXT

Ability to pass all context.

This allows the MQOO_PASS_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_PASS_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_IDENTITY_CONTEXT

Ability to set identity context.

This allows the MQOO_SET_IDENTITY_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_IDENTITY_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_SET_ALL_CONTEXT

Ability to set all context.

This allows the MQOO_SET_ALL_CONTEXT option to be specified on the MQOPEN call, and the MQPMO_SET_ALL_CONTEXT option to be specified on the MQPUT and MQPUT1 calls.

MQZAO_ALTERNATE_USER_AUTHORITY

Ability to use alternate user authority.

This allows the MQOO_ALTERNATE_USER_AUTHORITY option to be specified on the MQOPEN call, and the MQPMO_ALTERNATE_USER_AUTHORITY option to be specified on the MQPUT1 call.

MQZAO_ALL_MQI

All of the MQI authorizations.

This enables all of the authorizations described previously.

The following authorizations apply to administration of a queue manager:

MQZAO_CREATE

Ability to create objects of a specified type.

MQZAO_DELETE

Ability to delete a specified object.

MQZAO_DISPLAY

Ability to display the attributes of a specified object.

MQZAO_CHANGE

Ability to change the attributes of a specified object.

MQZAO_CLEAR

Ability to delete all messages from a specified queue.

MQZAO_AUTHORIZE

Ability to authorize other users for a specified object.

MQZAO_CONTROL

Ability to start, stop, or ping a non-client channel object.

MQZAO_CONTROL_EXTENDED

Ability to reset a sequence number, or resolve an indoubt message on a non-client channel object.

MQZAO_ALL_ADMIN

All of the administration authorizations, other than MQZAO_CREATE.

The following authorizations apply to both use of the MQI and to administration of a queue manager:

MQZAO_ALL

All authorizations, other than MQZAO_CREATE.

MQZAO_NONE

No authorizations.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_CHECK_AUTHORITY (QMgrName, EntityName, EntityType, ObjectName,
                    ObjectType, Authority, ComponentData,
                    &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR12  EntityName;         /* Entity name */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
```

```

MQLONG    ObjectType;          /* Object type */
MQLONG    Authority;          /* Authority to be checked */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */

```

MQZ_CHECK_PRIVILEGED - Check if user is privileged

This function is provided by an MQZAS_VERSION_6 authorization service component, and is invoked by the queue manager to determine whether a specified user is a privileged user.

The function identifier for this function (for MQZEP) is MQZID_CHECK_PRIVILEGED.

Syntax

```
MQZ_CHECK_PRIVILEGED( QMgrName , EntityData , EntityType , ComponentData ,
                     Continuation , CompCode , Reason )
```

Parameters

QMgrName

Type: MQCHAR48 - input

Queue manager name. The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

EntityData

Type: MQZED - input

Entity data. Data relating to the entity that is to be checked. For more information, see [“MQZED - Entity descriptor”](#) on page 1681.

EntityType

Type: MQLONG - input

Entity type. The type of entity specified by EntityData. It must be one of the following values:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ComponentData

Type: MQBYTE×ComponentDataLength - input/output

Component data. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of these component functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation

Type: MQLONG - output

Continuation indicator set by component. The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_CHECK_AUTHORITY, this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

If the call to a component fails (that is, *CompCode* returns MQCC_FAILED), and the *Continuation* parameter is MQZCI_DEFAULT or MQZCI_CONTINUE, the queue manager continues to call other components if there are any.

If the call succeeds (that is, *CompCode* returns MQCC_OK) no other components are called no matter what the setting of *Continuation* is.

If the call fails and the *Continuation* parameter is MQZCI_STOP then no other components are called and the error is returned to the queue manager. Components have no knowledge of previous calls, so the *Continuation* parameter is always set to MQZCI_DEFAULT before the call.

CompCode

Type: MQLONG - output

Completion code. It must be one of the following values:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason

Type: MQLONG - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_PRIVILEGED

(2584, X'A18') This user is not a privileged user ID.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information about these reason codes, see [API completion and reason codes](#).

C invocation

```
MQZ_CHECK_PRIVILEGED (QMgrName, &EntityData, EntityType,
                     ComponentData, &Continuation,
                     &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQZED     EntityData;         /* Entity name */
MQLONG    EntityType;         /* Entity type */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_COPY_ALL_AUTHORITY - Copy all authority

This function is provided by an authorization service component. It is invoked by the queue manager to copy all of the authorizations that are currently in force for a reference object to another object.

The function identifier for this function (for MQZEP) is MQZID_COPY_ALL_AUTHORITY.

Syntax

MQZ_COPY_ALL_AUTHORITY (*QMgrName, RefObjectName, ObjectName, ObjectType, ComponentData, Continuation, CompCode, Reason*)

Parameters

The MQZ_COPY_ALL_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

RefObjectName (MQCHAR48) - input

Reference object name.

The name of the reference object, the authorizations for which are to be copied. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

ObjectName (MQCHAR48) - input

Object name.

The name of the object for which accesses are to be set. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

ObjectType (MQLONG) - input

Object type.

The type of object specified by *RefObjectName* and *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_COPY_ALL_AUTHORITY this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_REF_OBJECT

(2294, X'8F6') Reference object unknown.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_COPY_ALL_AUTHORITY (QMgrName, RefObjectName, ObjectName, ObjectType,  
                        ComponentData, &Continuation, &CompCode,  
                        &Reason);
```

The parameters passed to the service are declared as follows:

```

MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR48  RefObjectName;      /* Reference object name */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */

```

MQZ_DELETE_AUTHORITY - Delete authority

This function is provided by an authorization service component, and is invoked by the queue manager to delete all of the authorizations associated with the specified object.

The function identifier for this function (for MQZEP) is MQZID_DELETE_AUTHORITY.

Syntax

MQZ_DELETE_AUTHORITY (*QMgrName*, *ObjectName*, *ObjectType*,
ComponentData, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_DELETE_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

ObjectName (MQCHAR48) - input

Object name.

The name of the object for which accesses are to be deleted. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType (MQLONG) - input

Object type.

The type of entity specified by *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_DELETE_AUTHORITY this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_DELETE_AUTHORITY (QMgrName, ObjectName, ObjectType, ComponentData,
                      &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48 QMgrName;           /* Queue manager name */
MQCHAR48 ObjectName;         /* Object name */
```

```

MQLONG    ObjectType;        /* Object type */
MQBYTE    ComponentData[n]; /* Component data */
MQLONG    Continuation;      /* Continuation indicator set by
                             component */
MQLONG    CompCode;          /* Completion code */
MQLONG    Reason;            /* Reason code qualifying CompCode */

```

MQZ_ENUMERATE_AUTHORITY_DATA - Enumerate authority data

This function is provided by an MQZAS_VERSION_4 authorization service component, and is invoked repeatedly by the queue manager to retrieve all of the authority data that matches the selection criteria specified on the first invocation.

The function identifier for this function (for MQZEP) is MQZID_ENUMERATE_AUTHORITY_DATA.

Syntax

MQZ_ENUMERATE_AUTHORITY_DATA (*QMgrName*, *StartEnumeration*, *Filter*, *AuthorityBufferLength*, *AuthorityBuffer*, *AuthorityDataLength*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_ENUMERATE_AUTHORITY_DATA call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

StartEnumeration (MQLONG) - input

Flag indicating whether call should start enumeration.

This indicates whether the call should start the enumeration of authority data, or continue the enumeration of authority data started by a previous call to MQZ_ENUMERATE_AUTHORITY_DATA. The value is one of the following:

MQZSE_START

Start enumeration.

The call is invoked with this value to start the enumeration of authority data. The *Filter* parameter specifies the selection criteria to be used to select the authority data returned by this and successive calls.

MQZSE_CONTINUE

Continue enumeration.

The call is invoked with this value to continue the enumeration of authority data. The *Filter* parameter is ignored in this case, and can be specified as the null pointer (the selection criteria are determined by the *Filter* parameter specified by the call that had *StartEnumeration* set to MQZSE_START).

Filter (MQZAD) - input

Filter.

If *StartEnumeration* is MQZSE_START, *Filter* specifies the selection criteria to be used to select the authority data to return. If *Filter* is the null pointer, no selection criteria are used, that is, all authority data is returned. See [“MQZAD \(Authority data\) on IBM i” on page 1719](#) for details of the selection criteria that can be used.

If *StartEnumeration* is MQZSE_CONTINUE, *Filter* is ignored, and can be specified as the null pointer.

AuthorityBufferLength (MQLONG) - input

Length of *AuthorityBuffer*.

This is the length in bytes of the *AuthorityBuffer* parameter. The authority buffer must be big enough to accommodate the data to be returned.

AuthorityBuffer (MQZAD) - output

Authority data.

This is the buffer in which the authority data is returned. The buffer must be big enough to accommodate an MQZAD structure, an MQZED structure, plus the longest entity name and longest domain name defined.

Note: This parameter is defined as an MQZAD, as the MQZAD always occurs at the start of the buffer. However, if the buffer is actually declared as an MQZAD, the buffer will be too small - it needs to be bigger than an MQZAD so that it can accommodate the MQZAD, MQZED, plus entity and domain names.

AuthorityDataLength (MQLONG) - output

Length of data returned in *AuthorityBuffer*.

This is the length of the data returned in *AuthorityBuffer*. If the authority buffer is too small, *AuthorityDataLength* is set to the length of the buffer required, and the call returns completion code MQCC_FAILED and reason code MQRC_BUFFER_LENGTH_ERROR.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_ENUMERATE_AUTHORITY_DATA this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_BUFFER_LENGTH_ERROR

(2005, X'7D5') Buffer length parameter not valid.

MQRC_NO_DATA_AVAILABLE

(2379, X'94B') No data available.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_ENUMERATE_AUTHORITY_DATA (QMgrName, StartEnumeration, &Filter,  
                               AuthorityBufferLength,  
                               &AuthorityBuffer,  
                               &AuthorityDataLength, ComponentData,  
                               &Continuation, &CompCode,  
                               &Reason);
```

The parameters passed to the service are declared as follows:

MQCHAR48	QMgrName;	/* Queue manager name */
MQLONG	StartEnumeration;	/* Flag indicating whether call should start enumeration */
MQZAD	Filter;	/* Filter */
MQLONG	AuthorityBufferLength;	/* Length of AuthorityBuffer */
MQZAD	AuthorityBuffer;	/* Authority data */
MQLONG	AuthorityDataLength;	/* Length of data returned in AuthorityBuffer */
MQBYTE	ComponentData[n];	/* Component data */
MQLONG	Continuation;	/* Continuation indicator set by component */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code qualifying CompCode */

MQZ_FREE_USER - Free user

This function is provided by a MQZAS_VERSION_5 authorization service component, and is invoked by the queue manager to free associated allocated resource. It is invoked when an application has finished running under all user contexts, for example during an MQDISC MQI call.

The function identifier for this function (for MQZEP) is MQZID_FREE_USER.

MQZ_GET_AUTHORITY - Get authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is invoked by the queue manager to retrieve the authority that an entity has to access the specified object.

The function identifier for this function (for MQZEP) is MQZID_GET_AUTHORITY.

Syntax

MQZ_GET_AUTHORITY (*QMgrName, EntityName, EntityType, ObjectName,*
ObjectType, Authority, ComponentData, Continuation, CompCode, Reason)

Parameters

The MQZ_GET_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

EntityName (MQCHAR12) - input

Entity name.

The name of the entity whose access to the object is to be retrieved. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType (MQLONG) - input

Entity type.

The type of entity specified by *EntityName*. The following value can be specified:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName (MQCHAR48) - input

Object name.

The name of the object for which the entity's authority is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType (MQLONG) - input

Object type.

The type of entity specified by *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority (MQLONG) - output

Authority of entity.

If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_GET_AUTHORITY this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_GET_AUTHORITY (QMgrName, EntityName, EntityType, ObjectName,  
                  ObjectType, &Authority, ComponentData,  
                  &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR12  EntityName;         /* Entity name */  
MQLONG    EntityType;         /* Entity type */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */
```

MQLONG	Authority;	/* Authority of entity */
MQBYTE	ComponentData[n];	/* Component data */
MQLONG	Continuation;	/* Continuation indicator set by component */
MQLONG	CompCode;	/* Completion code */
MQLONG	Reason;	/* Reason code qualifying CompCode */

MQZ_GET_EXPLICIT_AUTHORITY - Get explicit authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is invoked by the queue manager to retrieve the authority that a named group has to access a specified object (but without the additional authority of the **nobody** group), or the authority that the primary group of the named principal has to access a specified object.

The function identifier for this function (for MQZEP) is MQZID_GET_EXPLICIT_AUTHORITY.

Syntax

MQZ_GET_EXPLICIT_AUTHORITY (*QMgrName*, *EntityName*, *EntityType*, *ObjectName*, *ObjectType*, *Authority*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_GET_EXPLICIT_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

EntityName (MQCHAR12) - input

Entity name.

The name of the entity from which access to the object is to be retrieved. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType (MQLONG) - input

Entity type.

The type of entity specified by *EntityName*. The following value can be specified:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName (MQCHAR48) - input

Object name.

The name of the object for which the entity's authority is to be retrieved. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType (MQLONG) - input

Object type.

The type of entity specified by *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority (MQLONG) - output

Authority of entity.

If the entity has one authority, this field is equal to the appropriate authorization operation (MQZAO_* constant). If it has more than one authority, this field is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_GET_EXPLICIT_AUTHORITY this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_GET_EXPLICIT_AUTHORITY (QMgrName, EntityName, EntityType,  
                             ObjectName, ObjectType, &Authority,  
                             ComponentData, &Continuation,  
                             &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQCHAR12  EntityName;         /* Entity name */  
MQLONG    EntityType;         /* Entity type */  
MQCHAR48  ObjectName;         /* Object name */  
MQLONG    ObjectType;         /* Object type */  
MQLONG    Authority;          /* Authority of entity */  
MQBYTE    ComponentData[n];   /* Component data */  
MQLONG    Continuation;       /* Continuation indicator set by  
                               component */  
MQLONG    CompCode;           /* Completion code */  
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_INIT_AUTHORITY - Initialize authorization service

This function is provided by an authorization service component, and is invoked by the queue manager during configuration of the component. It is expected to call MQZEP in order to provide information to the queue manager.

The function identifier for this function (for MQZEP) is MQZID_INIT_AUTHORITY.

Syntax

MQZ_INIT_AUTHORITY (*Hconfig*, *Options*, *QMgrName*, *ComponentDataLength*,
ComponentData, *Version*, *CompCode*, *Reason*)

Parameters

The MQZ_INIT_AUTHORITY call has the following parameters.

Hconfig (MQHCONFIG) - input

Configuration handle.

This handle represents the particular component being initialized. It is to be used by the component when calling the queue manager with the MQZEP function.

Options (MQLONG) - input

Initialization options.

It is one of the following:

MQZIO_PRIMARY

Primary initialization.

MQZIO_SECONDARY

Secondary initialization.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

ComponentDataLength (MQLONG) - input

Length of component data.

Length in bytes of the *ComponentData* area. This length is defined in the component configuration data.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This is initialized to all zeros before calling the component's primary initialization function. This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions (including the initialization function) provided by this component are preserved, and presented the next time one of this component's functions is called.

Version (MQLONG) - input/output

Version number.

On input to the initialization function, this identifies the *highest* version number that the queue manager supports. The initialization function must change this, if necessary, to the version of the interface which *it* supports. If on return the queue manager does not support the version returned by the component, it calls the component's MQZ_TERM_AUTHORITY function and makes no further use of this component.

The following values are supported:

MQZAS_VERSION_1

Version 1.

MQZAS_VERSION_2

Version 2.

MQZAS_VERSION_3

Version 3.

MQZAS_VERSION_4

Version 4.

MQZAS_VERSION_5

Version 5.

MQZAS_VERSION_6

Version 6.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_INITIALIZATION_FAILED

(2286, X'8EE') Initialization failed for an undefined reason.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_INIT_AUTHORITY (Hconfig, Options, QMgrName, ComponentDataLength,  
                    ComponentData, &Version, &CompCode,  
                    &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG  Hconfig;           /* Configuration handle */  
MQLONG     Options;           /* Initialization options */  
MQCHAR48   QMgrName;         /* Queue manager name */  
MQLONG     ComponentDataLength; /* Length of component data */  
MQBYTE     ComponentData[n]; /* Component data */  
MQLONG     Version;          /* Version number */  
MQLONG     CompCode;         /* Completion code */  
MQLONG     Reason;           /* Reason code qualifying CompCode */
```

MQZ_INQUIRE - Inquire authorization service

This function is provided by a MQZAS_VERSION_5 authorization service component, and is invoked by the queue manager to query the supported functionality. Where multiple service components are used, service components are called in reverse order to the order they were installed in.

The function identifier for this function (for MQZEP) is MQZID_INQUIRE.

Syntax

MQZ_INQUIRE

(*QMgrName*, *SelectorCount*, *Selectors*, *IntAttrCount*, *IntAttrs*, *CharAttrLength*,
CharAttrs, *SelectorReturned*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_INQUIRE call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

SelectorCount (MQLONG) - input

Number of selectors.

The number of selectors supplied in the Selectors parameter.

The value must be between zero and 256.

Selectors (MQLONG x SelectorCount) - input

Selectors.

Array of selectors. Each selector identifies a required attribute and must be of one of the following types:

- MQIACF_* (integer)
- MQCACF_* (character)

Selectors can be specified in any order. The number of selectors in the array is indicated by the SelectorCount parameter.

Integer attributes identified by selectors are returned in the IntAttrs parameter in the same order as they appear in Selectors.

Character attributes identified by selectors are returned in the CharAttrs parameter in the same order as they appear in Selectors.

IntAttrCount (MQLONG) - input

Number of integer attributes.

The number of integer attributes supplied in the IntAttrs parameter.

The value must be in the range 0 through 256.

IntAttrs (MQLONG x IntAttrCount) - output

Integer attributes.

Array of integer attributes. The integer attributes are returned in the same order as the corresponding integer selectors in the Selectors array.

CharAttrCount (MQLONG) - input

Length of the character attributes buffer.

The length in bytes of the CharAttrs parameter.

The value must at least sum of the lengths of the requested character attributes. If no character attributes are requested, zero is a valid value.

CharAttrs (MQLONG x CharAttrCount) - output

Character attributes buffer.

Buffer containing character attributes, concatenated together. The character attributes are returned in the same order as the corresponding character selectors in the Selectors array.

The length of the buffer is given by the CharAttrCount parameter.

SelectorReturned (MQLONGxSelectorCount) - input

Selector returned.

Array of values identifying which attributes have been returned from the set requested for by the selectors in the Selectors parameter. The number of values in this array is indicated by the SelectorCount parameter. Each value in the array relates to the selector from the corresponding position in the Selectors array. Each value is one of the following:

MQZSL_RETURNED

The attribute requested by the corresponding selector in the Selectors parameter has been returned.

MQZSL_NOT_RETURNED

The attribute requested by the corresponding selector in the Selectors parameter has not been returned.

The array is initialized with all values as *MQZSL_NOT_RETURNED*. When an authorization service component returns an attribute, it sets the appropriate value in the array to *MQZSL_RETURNED*. This

allows any other authorization service components, to which the inquire call is made, to identify which attributes have already been returned.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation flag.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on other components.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_WARNING

Partial completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CHAR_ATTRS_TOO_SHORT

Not enough space for character attributes.

MQRC_INT_COUNT_TOO_SMALL

Not enough space for integer attributes.

If *CompCode* is MQCC_FAILED:

MQRC_SELECTOR_COUNT_ERROR

Number of selectors is not valid.

MQRC_SELECTOR_ERROR

Attribute selector not valid.

MQRC_SELECTOR_LIMIT_EXCEEDED

Too many selectors specified.

MQRC_INT_ATTR_COUNT_ERROR

Number of integer attributes is not valid.

MQRC_INT_ATTRS_ARRAY_ERROR

Integer attributes array not valid.

MQRC_CHAR_ATTR_LENGTH_ERROR

Number of character attributes is not valid.

MQRC_CHAR_ATTRS_ERROR

Character attributes string is not valid.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

C invocation

```
MQZ_INQUIRE (QMgrName, SelectorCount, Selectors, IntAttrCount,
              &IntAttrs, CharAttrLength, &CharAttrs,
              SelectorReturned, ComponentData, &Continuation,
              &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQLONG    SelectorCount;      /* Selector count */
MQLONG    Selectors[n];       /* Selectors */
MQLONG    IntAttrCount;       /* IntAttrs count */
MQLONG    IntAttrs[n];        /* Integer attributes */
MQLONG    CharAttrCount;      /* CharAttrs count */
MQLONG    CharAttrs[n];       /* Character attributes */
MQLONG    SelectorReturned[n]; /* Selector returned */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_REFRESH_CACHE - Refresh all authorizations

This function is provided by an MQZAS_VERSION_3 authorization service component. It is invoked by the queue manager to refresh the list of authorizations held internally by the component.

The function identifier for this function (for MQZEP) is MQZID_REFRESH_CACHE (8L).

Syntax

MQZ_REFRESH_CACHE

(QMgrName, ComponentData, Continuation, CompCode, Reason)

Parameters

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to use it in any defined manner.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component. Any changes made to it by any of the functions provided by this component are preserved, and presented the next time a function of the component is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_REFRESH_CACHE, this has the same effect as MQZCI_CONTINUE.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

C invocation

```
MQZ_REFRESH_CACHE (QMgrName, ComponentData,  
                  &Continuation, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */  
MQBYTE    ComponentData[n];  /* Component data */  
MQLONG    Continuation;      /* Continuation indicator set by  
                             component */  
MQLONG    CompCode;          /* Completion code */  
MQLONG    Reason;            /* Reason code qualifying CompCode */
```

MQZ_SET_AUTHORITY - Set authority

This function is provided by a MQZAS_VERSION_1 authorization service component, and is invoked by the queue manager to set the authority that an entity has to access the specified object.

The function identifier for this function (for MQZEP) is MQZID_SET_AUTHORITY.

Note: This function overrides any existing authorities. To preserve any existing authorities you must set them again with this function.

Syntax

MQZ_SET_AUTHORITY (*QMgrName*, *EntityName*, *EntityType*, *ObjectName*,
ObjectType, *Authority*, *ComponentData*, *Continuation*, *CompCode*, *Reason*)

Parameters

The MQZ_SET_AUTHORITY call has the following parameters.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

EntityName (MQCHAR12) - input

Entity name.

The name of the entity for which access to the object is to be set. The maximum length of the string is 12 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

EntityType (MQLONG) - input

Entity type.

The type of entity specified by *EntityName*. The following value can be specified:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

ObjectName (MQCHAR48) - input

Object name.

The name of the object to which access is required. The maximum length of the string is 48 characters; if it is shorter than that it is padded to the right with blanks. The name is not terminated by a null character.

If *ObjectType* is MQOT_Q_MGR, this name is the same as *QMgrName*.

ObjectType (MQLONG) - input

Object type.

The type of entity specified by *ObjectName*. It is one of the following:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority (MQLONG) - input

Authority to be checked.

If one authorization is being set, this field is equal to the appropriate authorization operation (MQZAO_* constant). If more than one authorization is being set, it is the bitwise OR of the corresponding MQZAO_* constants.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter of the MQZ_INIT_AUTHORITY call.

Continuation (MQLONG) - output

Continuation indicator set by component.

The following values can be specified:

MQZCI_DEFAULT

Continuation dependent on queue manager.

For MQZ_SET_AUTHORITY this has the same effect as MQZCI_STOP.

MQZCI_CONTINUE

Continue with next component.

MQZCI_STOP

Do not continue with next component.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_NOT_AUTHORIZED

(2035, X'7F3') Not authorized for access.

MQRC_SERVICE_ERROR

(2289, X'8F1') Unexpected error occurred accessing service.

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_UNKNOWN_ENTITY

(2292, X'8F4') Entity unknown to service.

C invocation

```
MQZ_SET_AUTHORITY (QMgrName, EntityName, EntityType, ObjectName,
                  ObjectType, Authority, ComponentData,
                  &Continuation, &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQCHAR48  QMgrName;           /* Queue manager name */
MQCHAR12  EntityName;         /* Entity name */
MQLONG    EntityType;         /* Entity type */
MQCHAR48  ObjectName;         /* Object name */
MQLONG    ObjectType;         /* Object type */
MQLONG    Authority;          /* Authority to be checked */
MQBYTE    ComponentData[n];   /* Component data */
MQLONG    Continuation;       /* Continuation indicator set by
                               component */
MQLONG    CompCode;           /* Completion code */
MQLONG    Reason;             /* Reason code qualifying CompCode */
```

MQZ_TERM_AUTHORITY - Terminate authorization service

This function is provided by an authorization service component, and is invoked by the queue manager when it no longer requires the services of this component. The function must perform any cleanup required by the component.

The function identifier for this function (for MQZEP) is MQZID_TERM_AUTHORITY.

Syntax

MQZ_TERM_AUTHORITY (*Hconfig, Options, QMgrName, ComponentData, CompCode, Reason*)

Parameters

The MQZ_TERM_AUTHORITY call has the following parameters.

Hconfig (MQHCONFIG) - input

Configuration handle.

This handle represents the particular component being terminated.

Options (MQLONG) - input

Termination options.

It is one of the following:

MQZTO_PRIMARY

Primary termination.

MQZTO_SECONDARY

Secondary termination.

QMgrName (MQCHAR48) - input

Queue manager name.

The name of the queue manager calling the component. This name is padded with blanks to the full length of the parameter; the name is not terminated by a null character.

The queue-manager name is passed to the component for information; the authorization service interface does not require the component to make use of it in any defined manner.

ComponentData (MQBYTE x ComponentDataLength) - input/output

Component data.

This data is kept by the queue manager on behalf of this particular component; any changes made to it by any of the functions provided by this component are preserved, and presented the next time one of this component's functions is called.

The length of this data area is passed by the queue manager in the *ComponentDataLength* parameter on the MQZ_INIT_AUTHORITY call.

When the MQZ_TERM_AUTHORITY call has completed, the queue manager discards this data.

CompCode (MQLONG) - output

Completion code.

It is one of the following:

MQCC_OK

Successful completion.

MQCC_FAILED

Call failed.

Reason (MQLONG) - output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE

(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_SERVICE_NOT_AVAILABLE

(2285, X'8ED') Underlying service not available.

MQRC_TERMINATION_FAILED

(2287, X'8FF') Termination failed for an undefined reason.

For more information on these reason codes, see [Messages and reason codes](#).

C invocation

```
MQZ_TERM_AUTHORITY (Hconfig, Options, QMgrName, ComponentData,  
                    &CompCode, &Reason);
```

The parameters passed to the service are declared as follows:

```
MQHCONFIG  Hconfig;          /* Configuration handle */  
MQLONG     Options;          /* Termination options */  
MQCHAR48   QMgrName;        /* Queue manager name */  
MQBYTE     ComponentData[n]; /* Component data */  
MQLONG     CompCode;         /* Completion code */  
MQLONG     Reason;           /* Reason code qualifying CompCode */
```

MQZAC - Application context

This parameter specifies data related to the calling application.

The MQZAC structure is used on the MQZ_AUTHENTICATE_USER call for the *ApplicationContext* parameter.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value is:

MQZAC_STRUC_ID

Identifier for application context structure.

For the C programming language, the constant MQZAC_STRUC_ID_ARRAY is also defined; this has the same value as MQZAC_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the service.

Version (MQLONG)

Structure version number.

The value is:

MQZAC_VERSION_1

Version-1 application context structure.

The following constant specifies the version number of the current version:

MQZAC_CURRENT_VERSION

Current version of application context structure.

This is an input field to the service.

ProcessId (MQPID)

Process identifier.

The process identifier of the application.

ThreadId (MQTID)

Thread identifier.

The thread identifier of the application.

ApplName (MQCHAR28)

Application name.

The application name.

UserID (MQCHAR12)

User identifier.

For IBM i systems the user profile that the application job was created under. (IBM i: when a profile swap is done with the QWTSETP API in the application job the current user profile is returned).

EffectiveUserID (MQCHAR12)

Effective user identifier.

For IBM i systems the application job's current user profile.

Environment (MQLONG)

Environment.

This field specifies the environment from which the call was made.

This can have one of the following values:

MQXE_COMMAND_SERVER

Command server.

MQXE_MQSC

runmqsc command interpreter.

MQXE_MCA

Message channel agent

MQXE_OTHER

Undefined environment

CallerType (MQLONG)

Caller Type.

This field specifies the type of program that made the call.

This can have one of the following values:

MQXACT_EXTERNAL

The call is external to the queue manager.

MQXACT_INTERNAL

The call is internal to the queue manager.

AuthenticationType (MQLONG)

Authentication Type.

This field specifies the type of authentication being performed.

This can have one of the following values:

MQZAT_INITIAL_CONTEXT

The authentication call is due to user context being initialized. This value is used during an MQCONN or MQCONNEX call.

MQZAT_CHANGE_CONTEXT

The authentication call is due to the user context being changed. This value is used when the MCA changes the user context.

v

BindType (MQLONG)

Bind Type.

This field specifies the type of binding in use.

This can have one of the following values:

MQCNO_FASTPATH_BINDING

Fastpath binding.

MQCNO_SHARED_BINDING

Shared binding.

MQCNO_ISOLATED_BINDING

Isolated binding.

C declaration

```
typedef struct tagMQZAC MQZAC;
struct tagMQZAC {
    MQCHAR4    StrucId;           /* Structure identifier */
    MQLONG     Version;           /* Structure version number */
    MQPID      ProcessId;         /* Process identifier */
    MQTID      ThreadId;          /* Thread identifier */
    MQCHAR28   ApplName;          /* Application name */
    MQCHAR12   UserID;            /* User identifier */
    MQCHAR12   EffectiveUserID;    /* Effective user identifier */
    MQLONG     Environment;        /* Environment */
    MQLONG     CallerType;         /* Caller type */
    MQLONG     AuthenticationType; /* Authentication type */
    MQLONG     BindType;           /* Bind type */
};
```

IBM i MQZAD (Authority data) on IBM i

The MQZAD structure is used on the MQZ_ENUMERATE_AUTHORITY_DATA call for two parameters.

The two parameters are:

- MQZAD is used for the **Filter** parameter which is input to the call. This parameter specifies the selection criteria that are to be used to select the authority data returned by the call.
- MQZAD is also used for the **AuthorityBuffer** parameter which is output from the call. This parameter specifies the authorizations for one combination of profile name, object type, and entity.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value is:

MQZAD_STRUC_ID

Identifier for authority data structure.

For the C programming language, the constant MQZAD_STRUC_ID_ARRAY is also defined; this has the same value as MQZAD_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the service.

Version (MQLONG)

Structure version number.

The value is:

MQZAD_VERSION_1

Version-1 authority data structure.

The following constant specifies the version number of the current version:

MQZAD_CURRENT_VERSION

Current version of authority data structure.

This is an input field to the service.

ProfileName (MQCHAR48)

Profile name.

For the **Filter** parameter, this field is the profile name from which authority data is required. If the name is entirely blank up to the end of the field or the first null character, authority data for all profile names is returned.

For the **AuthorityBuffer** parameter, this field is the name of a profile that matches the specified selection criteria.

ObjectType (MQLONG)

Object type.

For the **Filter** parameter, this field is the object type for which authority data is required. If the value is MQOT_ALL, authority data for all object types is returned.

For the **AuthorityBuffer** parameter, this field is the object type to which the profile identified by **ProfileName** applies.

The value is one of the following; for the **Filter** parameter, the value MQOT_ALL is also valid:

MQOT_AUTH_INFO

Authentication information.

MQOT_CHANNEL

Channel.

MQOT_CLNTCONN_CHANNEL

Client connection channel.

MQOT_LISTENER

Listener.

MQOT_NAMELIST

Namelist.

MQOT_PROCESS

Process definition.

MQOT_Q

Queue.

MQOT_Q_MGR

Queue manager.

MQOT_SERVICE

Service.

Authority (MQLONG)

Authority.

For the **Filter** parameter, this field is ignored.

For the **AuthorityBuffer** parameter, this field represents the authorizations that the entity has to the objects identified by **ProfileName** and **ObjectType**. If the entity has only one authority, the field is equal to the appropriate authorization value (MQZAO_* constant). If the entity has more than one authority, the field is the bitwise OR of the corresponding MQZAO_* constants.

EntityDataPtr (PMQZED)

Address of MQZED structure identifying an entity.

For the **Filter** parameter, this field points to an MQZED structure that identifies the entity from which authority data is required. If **EntityDataPtr** is the null pointer, authority data for all entities is returned.

For the **AuthorityBuffer** parameter, this field points to an MQZED structure that identifies the entity that the returned authority data came from.

EntityType (MQLONG)

Entity type.

For the **Filter** parameter, this field specifies the entity type for which authority data is required. If the value is MQZAET_NONE, authority data for all entity types is returned.

For the **AuthorityBuffer** parameter, this field specifies the type of the entity identified by the MQZED structure pointed to by **EntityDataPtr**.

The value is one of the following; for the **Filter** parameter, the value MQZAET_NONE is also valid:

MQZAET_PRINCIPAL

Principal.

MQZAET_GROUP

Group.

Options (MQAUTHOPT)

Options.

This field specifies options that give control over the profiles that are displayed.

One of the following must be specified:

MQAUTHOPT_NAME_ALL_MATCHING

Displays all profiles

MQAUTHOPT_NAME_EXPLICIT

Displays profiles that have exactly the same name as specified in the **ProfileName** field.

In addition, one of the following must also be specified:

MQAUTHOPT_ENTITY_SET

Display all profiles used to calculate the cumulative authority that the entity has to the object specified by **ProfileName**. The **ProfileName** field must not contain any wildcard characters.

- If the specified entity is a principal, for each member of the set {entity, groups} the most applicable profile that applies to the object is displayed.
- If the specified entity is a group, the most applicable profile from the group that applies to the object is displayed.
- If this value is specified, then the values of **ProfileName**, **ObjectType**, **EntityType**, and the entity name specified in the **EntityDataPtr** MQZED structure, must all be non-blank.

If you have specified *MQAUTHOPT_NAME_ALL_MATCHING*, you can also specify the following:

MQAUTHOPT_ENTITY_EXPLICIT

Displays profiles that have exactly the same entity name as the entity name specified in the **EntityDataPtr** MQZED structure.

C declaration

```
typedef struct tagMQZAD MQZAD;  
struct tagMQZAD {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    MQCHAR48    ProfileName;     /* Profile name */  
    MQLONG     ObjectType;       /* Object type */  
    MQLONG     Authority;        /* Authority */  
    PMQZED     EntityDataPtr;    /* Address of MQZED structure identifying an  
                                entity */  
    MQLONG     EntityType;       /* Entity type */  
    MQAUTHOPT  Options;         /* Options */  
};
```

MQZED - Entity descriptor

The MQZED structure is used in a number of authorization service calls to specify the entity for which authorization is to be checked.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value is:

MQZED_STRUC_ID

Identifier for entity descriptor structure.

For the C programming language, the constant MQZED_STRUC_ID_ARRAY is also defined; this has the same value as MQZED_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the service.

Version (MQLONG)

Structure version number.

The value is:

MQZED_VERSION_1

Version-1 entity descriptor structure.

The following constant specifies the version number of the current version:

MQZED_CURRENT_VERSION

Current version of entity descriptor structure.

This is an input field to the service.

EntityNamePtr (PMQCHAR)

Address of entity name.

This is a pointer to the name of the entity whose authorization is to be checked.

EntityDomainPtr (PMQCHAR)

Address of entity domain name.

This is a pointer to the name of the domain containing the definition of the entity whose authorization is to be checked.

SecurityId (MQBYTE40)

Security identifier.

This is the security identifier whose authorization is to be checked.

CorrelationPtr (MQPTR)

Correlation pointer.

This facilitates the passing of correlational data between the authenticate user function and other appropriate OAM functions.

C declaration

```
typedef struct tagMQZED MQZED;  
struct tagMQZED {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    PMQCHAR    EntityNamePtr;    /* Address of entity name */  
    PMQCHAR    EntityDomainPtr;  /* Address of entity domain name */  
    MQBYTE40   SecurityId;       /* Security identifier */  
    MQPTR      CorrelationPtr;   /* Address of correlation data */  
};
```

MQZFP - Free parameters

This parameter specifies data related to resource to be freed.

The MQZFP structure is used on the MQZ_FREE_USER call for the *FreeParms* parameter.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value is:

MQZFP_STRUC_ID

Identifier for free parameters structure.

For the C programming language, the constant MQZFP_STRUC_ID_ARRAY is also defined; this has the same value as MQZFP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the service.

Version (MQLONG)

Structure version number.

The value is:

MQZFP_VERSION_1

Version-1 free parameters structure.

The following constant specifies the version number of the current version:

MQZFP_CURRENT_VERSION

Current version of free parameters structure.

This is an input field to the service.

Reserved (MQBYTE8)

Reserved field.

The initial value is null.

CorrelationPtr (MQPTR)

Correlation pointer.

Address of correlation data relating to the resource to be freed.

C declaration

```
typedef struct tagMQZFP MQZFP;  
struct tagMQZFP {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    MQBYTE8    Reserved;         /* Reserved field */  
};
```

```
MQPTR    CorrelationPtr;    /* Address of correlation data */
};
```

MQZIC - Identity context

The MQZIC structure is used on the MQZ_AUTHENTICATE_USER call for the *IdentityContext* parameter.

The MQZIC structure contains identity context information, that identifies the user of the application that first put the message on a queue:

- The queue manager fills the *UserIdentifier* field with a name that identifies the user, the way that the queue manager can do this depends on the environment in which the application is running.
- The queue manager fills the *AccountingToken* field with a token or number that it determined from the application that put the message.
- Applications can use the *ApplIdentityData* field for any extra information that they want to include about the user (for example, an encrypted password).

Suitably authorized applications may set the identity context using the MQZ_AUTHENTICATE_USER function.

A Windows systems security identifier (SID) is stored in the *AccountingToken* field when a message is created under IBM MQ for Windows. The SID can be used to supplement the *UserIdentifier* field and to establish the credentials of a user.

Fields

StrucId (MQCHAR4)

Structure identifier.

The value is:

MQZIC_STRUC_ID

Identifier for identity context structure.

For the C programming language, the constant MQZIC_STRUC_ID_ARRAY is also defined; this has the same value as MQZIC_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the service.

Version (MQLONG)

Structure version number.

The value is:

MQZIC_VERSION_1

Version-1 identity context structure.

The following constant specifies the version number of the current version:

MQZIC_CURRENT_VERSION

Current version of identity context structure.

This is an input field to the service.

UserIdentifier (MQCHAR12)

User identifier.

This is part of the **identity context** of the message.

UserIdentifier specifies the user identifier of the application that originated the message. The queue manager treats this information as character data, but does not define the format of it. For more information on the *UserIdentifier* field, see [“UserIdentifier \(MQCHAR12\)” on page 441](#).

AccountingToken (MQBYTE32)

Accounting token.

This is part of the **identity context** of the message.

AccountingToken allows an application to cause work done as a result of the message to be appropriately charged. The queue manager treats this information as a string of bits and does not check its content. For more information on the *AccountingToken* field, see “*AccountingToken* (MQBYTE32)” on page 400.

ApplIdentityData (MQCHAR32)

Application data relating to identity.

This is part of the **identity context** of the message.

ApplIdentityData is information that is defined by the application suite that can be used to provide additional information about the origin of the message. For example, it could be set by applications running with suitable user authority to indicate whether the identity data is trusted. For more information on the *ApplIdentityData* field, see “*ApplIdentityData* (MQCHAR32)” on page 401.

C declaration

```
typedef struct tagMQZED MQZED;  
struct tagMQZED {  
    MQCHAR4    StrucId;           /* Structure identifier */  
    MQLONG     Version;          /* Structure version number */  
    MQCHAR12   UserIdentifier;    /* User identifier */  
    MQBYTE32   AccountingToken;  /* Accounting token */  
    MQCHAR32   ApplIdentityData; /* Application data relating to identity */  
};
```

Reference material for IBM MQ bridge for HTTP

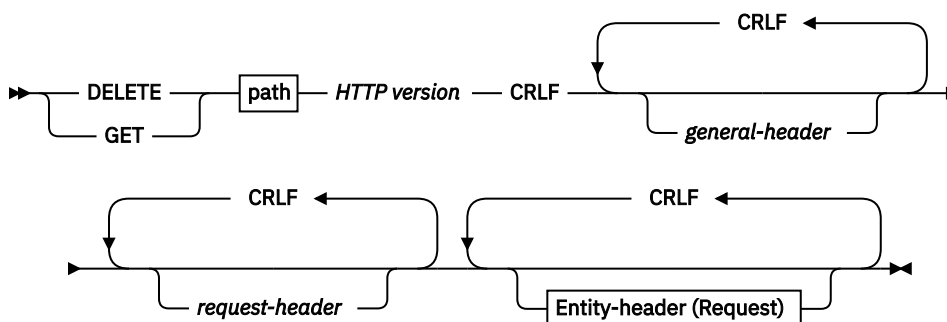
Reference topics for IBM MQ bridge for HTTP, arranged alphabetically

HTTP DELETE: IBM MQ bridge for HTTP command

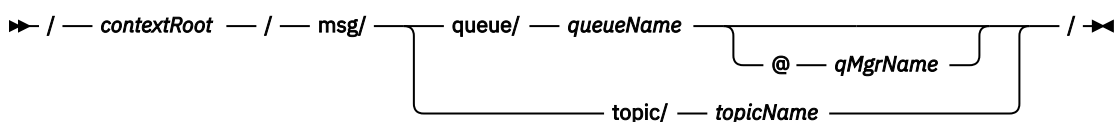
The HTTP **DELETE** operation gets a message from an IBM MQ queue, or retrieves a publication from a topic. The message is removed from the queue. If the publication is retained, it is not removed. A response message is sent back to the client including information about the message.

Syntax

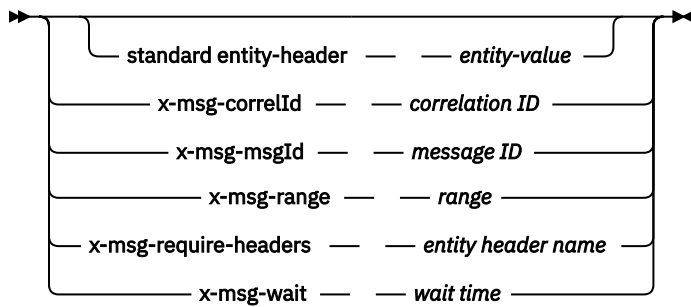
Request



Path



entity-header (Request)

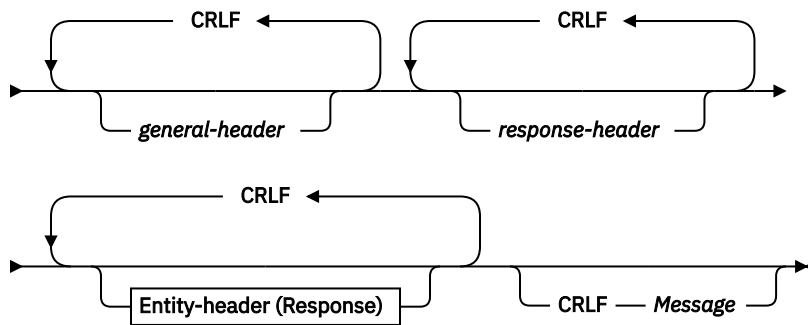


Note:

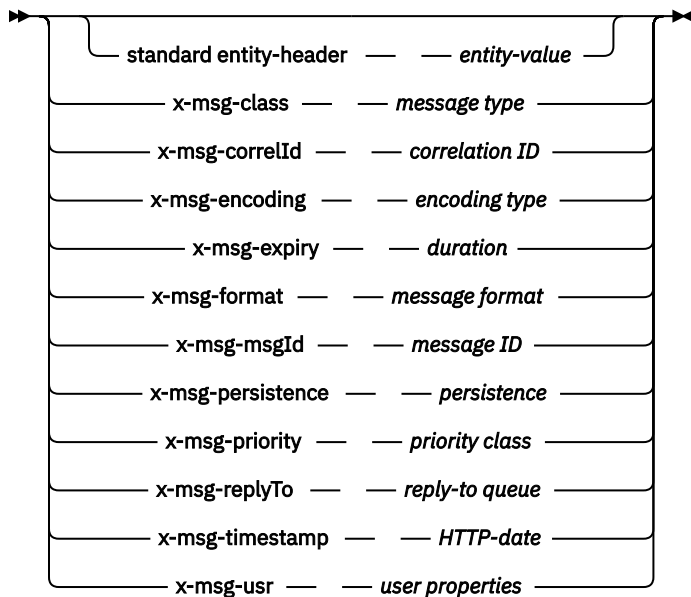
1. If a question mark (?) is used it must be substituted with %3f. For example, orange?topic should be specified as orange%3ftopic.
2. @ qMgrName is only valid on an HTTP **POST**

Response

➡ HTTP version — HTTP Status-Code — HTTP Reason-Phrase — CRLF ➡



entity-header (Response)



Request parameters

Path

See [“URI Format”](#) on page 1759.

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

request-header

See [HTTP/1.1 - 5.3 Request Header Fields](#). The Host field is mandatory on an HTTP/1.1 request. It is often automatically inserted by the tool you use to create a client request.

entity-header (Request)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity headers listed in the Request syntax diagram.

Response parameters**Path**

See [“URI Format” on page 1759](#).

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

response-header

See [HTTP/1.1 - 6.2 Response Header Fields](#).

entity-header (Response)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity or response headers listed in the Response syntax diagram. The Content-Length is always present in a response. It is set to zero if there is no message body.

Message

Message body.

Description

If the HTTP **DELETE** request is successful, the response message contains the data retrieved from the IBM MQ queue. The number of bytes in the body of the message is returned in the HTTP Content-Length header. The status code for the HTTP response is set to 200 OK. If x-msg-range is specified as 0, or 0-0, then the status code of the HTTP response is 204 No Content.

If the HTTP **DELETE** request is unsuccessful, the response includes an IBM MQ bridge for HTTP error message and an HTTP status code.

HTTP DELETE example

HTTP **DELETE** gets a message from a queue and deletes the message, or retrieves and deletes a publication. The **HTTPDELETE** Java sample is an example an HTTP **DELETE** request reading a message from a queue. Instead of using Java, you could create an HTTP **DELETE** request using a browser form, or an AJAX toolkit instead.

The following figure shows an HTTP request to delete the next message on queue called myQueue. In response, the message body is returned to the client. In IBM MQ terms, HTTP **DELETE** is a destructive get.

The request contains the HTTP request header x-msg-wait, which instructs IBM MQ bridge for HTTP how long to wait for a message to arrive on the queue. The request also contains the x-msg-require-headers request header, which specifies that the client is to receive the message correlation ID in the response.

```
DELETE /msg/queue/myQueue/ HTTP/1.1
Host: www.example.org
x-msg-wait: 10
x-msg-require-headers: correlID
```

Figure 37. Example of an HTTP **DELETE** request

The following figure shows the response returned to the client. The correlation ID is returned to the client, as requested in x-msg-require-headers of the request.

```
HTTP/1.1 200 OK
Date: Wed, 2 Jan 2007 22:38:34 GMT
Server: Apache-Coyote/1.1 WMQ-HTTP/1.1 JEE-Bridge/1.1
Content-Length: 50
Content-Type: text/plain; charset=utf-8
x-msg-correlId: 1234567890
```

Here is my message body that is retrieved from the queue.

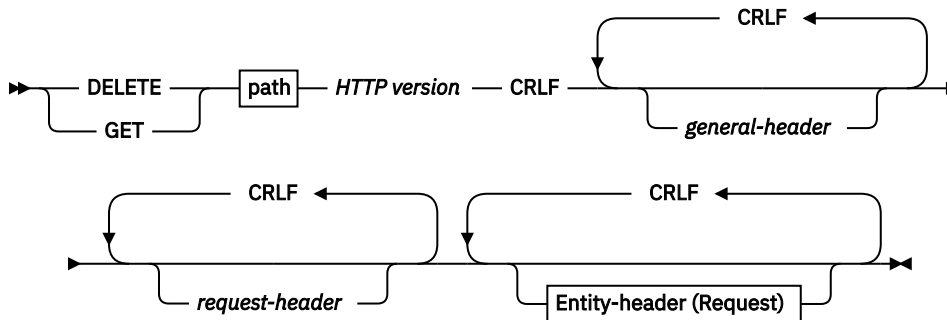
Figure 38. Example of an HTTP **DELETE** response

HTTP GET: IBM MQ bridge for HTTP command

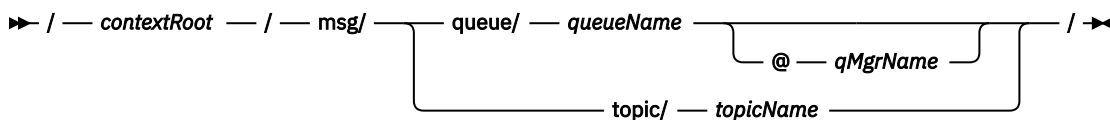
The HTTP **GET** operation gets a message from an IBM MQ queue. The message remains on the queue. The HTTP **GET** operation is equivalent to browsing an IBM MQ queue.

Syntax

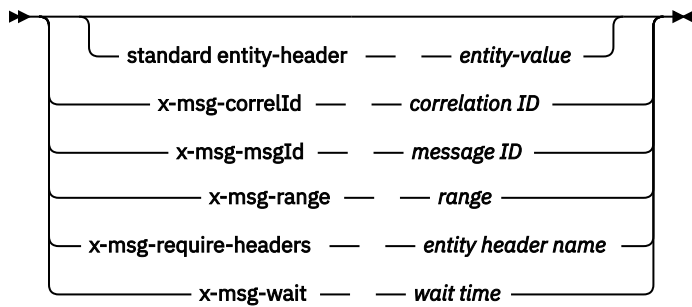
Request



Path



entity-header (Request)

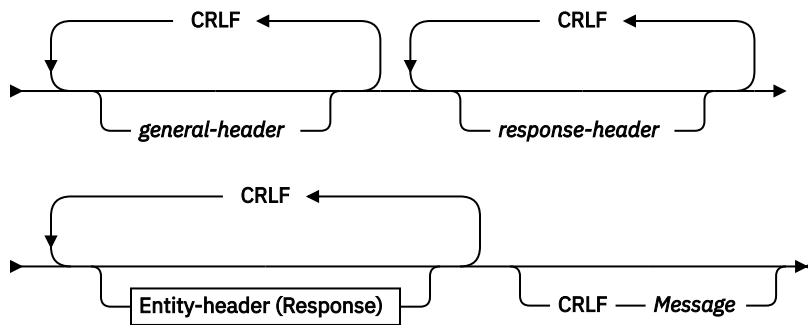


Note:

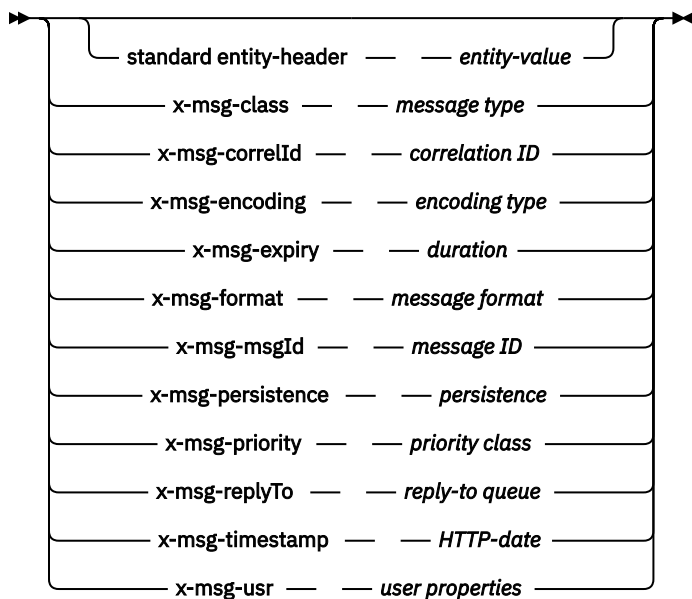
1. If a question mark (?) is used it must be substituted with %3f. For example, orange?topic should be specified as orange%3ftopic.
2. @ qMgrName is only valid on an HTTP **POST**

Response

➡ HTTP version — HTTP Status-Code — HTTP Reason-Phrase — CRLF ➡



entity-header (Response)



Request parameters

Path

See [“URI Format”](#) on page 1759.

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

request-header

See [HTTP/1.1 - 5.3 Request Header Fields](#). The Host field is mandatory on an HTTP/1.1 request. It is often automatically inserted by the tool you use to create a client request.

entity-header (Request)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity headers listed in the Request syntax diagram.

Response parameters**Path**

See [“URI Format” on page 1759](#).

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

response-header

See [HTTP/1.1 - 6.2 Response Header Fields](#).

entity-header (Response)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity or response headers listed in the Response syntax diagram. The Content-Length is always present in a response. It is set to zero if there is no message body.

Message

Message body.

Description

If the HTTP **GET** request is successful, the response message contains the data retrieved from the IBM MQ queue. The number of bytes in the body of the message is returned in the HTTP Content-Length header. The status code for the HTTP response is set to 200 OK. If x-msg-range is specified as 0, or 0-0, then the status code of the HTTP response is 204 No Content.

If the HTTP **GET** request is unsuccessful, the response includes an IBM MQ bridge for HTTP error message and an HTTP status code.

HTTP GET example

HTTP **GET** gets a message from a queue. The message remains on the queue. In IBM MQ terms, HTTP **GET** is a browse request. You could create an HTTP **GET** request using a Java client, a browser form, or an AJAX toolkit.

The following figure shows an HTTP request to browse the next message on queue called myQueue.

The request contains the HTTP request header x-msg-wait, which instructs IBM MQ bridge for HTTP how long to wait for a message to arrive on the queue. The request also contains the x-msg-require-headers request header, which specifies that the client is to receive the message correlation ID in the response.

```
GET /msg/queue/myQueue/ HTTP/1.1
Host: www.example.org
x-msg-wait: 10
x-msg-require-headers: correlID
```

Figure 39. Example of an HTTP **GET** request

The following figure shows the response returned to the client. The correlation ID is returned to the client, as requested in `x-msg-require-headers` of the request.

```
HTTP/1.1 200 OK
Date: Wed, 2 Jan 2007 22:38:34 GMT
Server: Apache-Coyote/1.1 WMQ-HTTP/1.1 JEE-Bridge/1.1
Content-Length: 50
Content-Type: text/plain; charset=utf-8
x-msg-correlId: 1234567890
```

Here is my message body that appears on the queue.

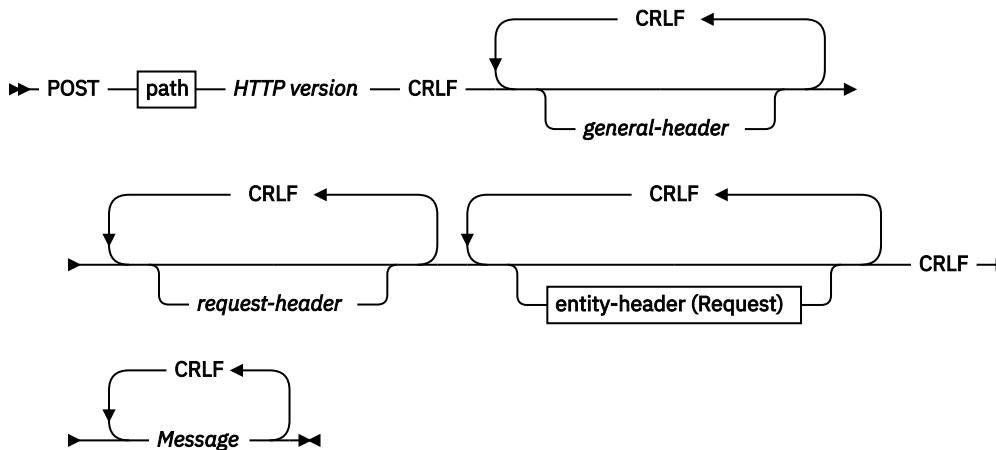
Figure 40. Example of an HTTP **GET** response

HTTP POST: IBM MQ bridge for HTTP command

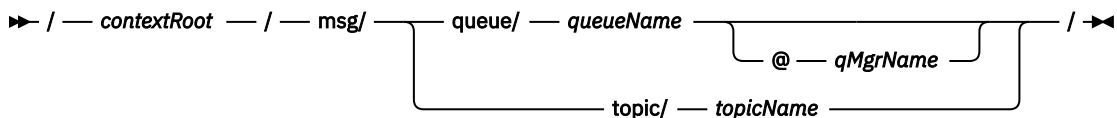
The HTTP **POST** operation puts a message on an IBM MQ queue, or publishes a message to a topic.

Syntax

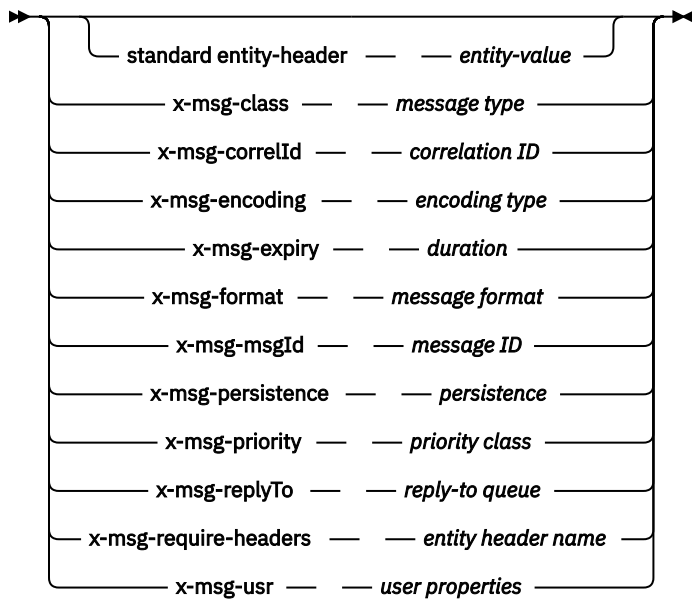
Request



Path



entity-header (Request)

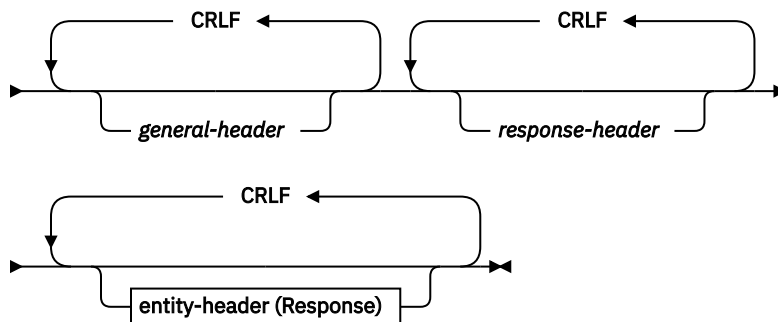


Note:

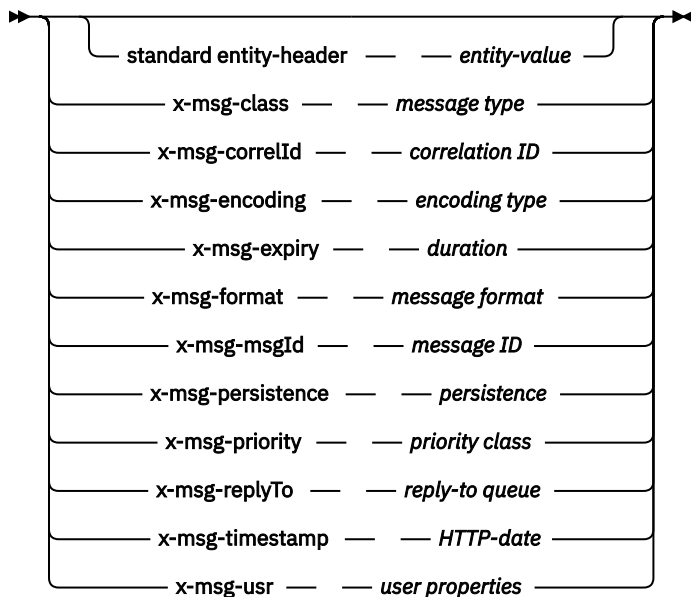
1. If a question mark (?) is used it must be substituted with %3f. For example, orange?topic should be specified as orange%3ftopic.
2. @ *qMgrName* is only valid on an HTTP **POST**

Response

➡ HTTP version — HTTP Status-Code — HTTP Reason-Phrase — CRLF ➡



entity-header (Response)



Request parameters

Path

See [“URI Format” on page 1759](#).

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

request-header

See [HTTP/1.1 - 5.3 Request Header Fields](#). The Host field is mandatory on an HTTP/1.1 request. It is often automatically inserted by the tool you use to create a client request.

entity-header (Request)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity headers listed in the Request syntax diagram. The Content-Length and Content-Type should be inserted in a request, and are often inserted automatically by the tool you use to create a client request. The Content-Type must match the type defined in the x-msg-class custom entity-header, if it is specified.

Message

Message to put onto the queue, or publication to post to a topic.

Response parameters

Path

See [“URI Format” on page 1759](#).

HTTP version

HTTP version; for example, HTTP/1.1

general-header

See [HTTP/1.1 - 4.5 General Header Fields](#).

response-header

See [HTTP/1.1 - 6.2 Response Header Fields](#).

entity-header (Response)

See [HTTP/1.1 - 7.1 Entity Header Fields](#). One of the entity or response headers listed in the Response syntax diagram. The Content-Length is always present in a response. It is set to zero if there is no message body.

Description

If no `x-msg-usr` header is included, and message class is BYTES or TEXT, the message put on the queue does not have an MQRFH2.

Use HTTP entity and request headers in the HTTP **POST** request to set the properties of the message that is put onto the queue. You can also use `x-msg-require-headers` to request which headers are returned in the response message.

If the HTTP **POST** request is successful, the entity of the response message is empty and its Content-Length is zero. The HTTP status code is 200 OK.

If the HTTP **POST** request is unsuccessful, the response includes an IBM MQ bridge for HTTP error message and an HTTP status code. The IBM MQ message is not put on the queue or topic.

HTTP POST example

HTTP **POST** puts a message to a queue, or a publication to a topic. The **HTTPPOST** Java sample is an example an HTTP **POST** request of a message to a queue. Instead of using Java, you could create an HTTP **POST** request using a browser form, or an AJAX toolkit instead.

The following figure shows an HTTP request to put a message on a queue called myQueue. This request contains the HTTP header `x-msg-correlID` to set the correlation ID of the IBM MQ message.

```
POST /msg/queue/myQueue/ HTTP/1.1
Host: www.example.org
Content-Type: text/plain
x-msg-correlID: 1234567890
Content-Length: 50

Here is my message body that is posted on the queue.
```

*Figure 41. Example of an HTTP **POST** request to a queue*

The following figure shows the response sent back to the client. There is no response content.

```
HTTP/1.1 200 OK
Date: Wed, 2 Jan 2007 22:38:34 GMT
Server: Apache-Coyote/1.1 WMQ-HTTP/1.1 JEE-Bridge/1.1
Content-Length: 0
```

*Figure 42. Example of an HTTP **POST** response*

HTTP headers

The IBM MQ bridge for HTTP supports custom request HTTP headers, custom entity HTTP headers, and a subset of standard HTTP headers.

HTTP practice is to prefix all custom headers with `x-`, so the IBM MQ Bridge for HTTP headers are prefixed with `x-msg-`. For example, to set the priority header use `x-msg-priority`.

Note:

- Most header values are case sensitive. For example, when using the `msgId` header, `NONE` is a keyword, whereas `none` is a `msgID`.
- Misspelled headers are ignored.

Custom entity HTTP headers

The custom entity HTTP headers contain information about IBM MQ messages. Using entity headers, you can set values in the message descriptor (MQMD), or query values in the MQMD. An additional entity header, `x-msg-usr`, sets and returns any user property information you want to associate with a request.

You can use entity headers in different HTTP request contexts:

DELETE

You can only use the `x-msg-correlId`, or `x-msg-msgId`, or both, entity headers with a **DELETE** HTTP request. The effect of the headers is to select a particular message by `MsgId` and `CorrelId` in an `MQGET`, and to delete the message from its queue.

GET

You can only use the `x-msg-correlId`, or `x-msg-msgId`, or both, entity headers with a **GET** HTTP request. The effect of the headers is to select a particular message by `MsgId` and `CorrelId` in an `MQGET` for browse.

POST

You can use any entity header in a **POST** HTTP request, except `x-msg-timestamp`.

x-msg-require-headers

On any **GET**, **POST** or **DELETE** HTTP request, you can add multiple entity headers inside the `x-msg-require-headers` request header, separated by commas. The effect is to return the specified entity headers in the HTTP response message, containing the value of the associated message property.

The description of each header lists in which contexts the header is processed by IBM MQ bridge for HTTP. For example, in the header **POST**, `x-msg-require-headers`, the header is processed by IBM MQ bridge for HTTP in an HTTP **POST** request, or in the `x-msg-require-headers` request header in either an HTTP **POST**, **GET**, or **DELETE** request. If the header is included in a context it is not allowed in, the header is ignored. No error is reported.

You can put any standard HTTP headers into requests to be processed by the Web server, or other request handlers. Similarly, the response might contain other standard HTTP headers inserted by the Web server or other response handlers.

Custom request HTTP headers

The three custom request HTTP headers, `x-msg-range`, `x-msg-require-headers`, and `x-msg-wait`, pass additional information about the HTTP request to the server. They act as request modifiers. With `x-msg-range`, you can restrict the amount of message data returned in a response. With `x-msg-require-headers`, you can request the response to contain information about the result of the request. With `x-msg-wait`, you can modify the time the client waits for an HTTP response.

Standard HTTP headers

The `Host` standard HTTP request-header must be specified in an HTTP/1.1 request.

The `Content-Length` and `Content-Type` standard HTTP entity headers can be specified in a request.

The `Content-Length`, `Content-Location`, `Content-Range`, `Content-Type`, and `Server` standard HTTP entity headers can be returned in response to a request. Specify one or more of the standard HTTP headers in the `x-msg-request-header` header in the request message.

Alphabetic list of HTTP headers

class: HTTP x-msg-class entity-header

Set or return the message type.

Type	Description
HTTP header name	<code>x-msg-class</code>

Type	Description
HTTP header type	Entity-header
Valid in HTTP request message	POST, x-msg-require-headers
Allowed values	BYTES MAP STREAM TEXT
Default value	BYTES

Description

- In an HTTP **POST** request, sets the type of the message created.
- Specifying the class header on a **GET** or **DELETE** returns a 400 Bad Request with entity body of MQHTTP40007.
- Specified in x-msg-require-headers, sets x-msg-class in the HTTP response message to the type of a message.
- If an invalid value is specified for this header a MQHTTP40005 message is returned.
- If the x-msg-class header is not specified and the content-type of the message is application/x-www-form-urlencoded, the data is assumed to be a JMS map object.

Content-Length: HTTP entity-header

Set or return the length, in bytes, of the body of the message.

Type	Description
HTTP header name	Content-Length
HTTP header type	Entity-header
Valid in HTTP request message	x-msg-require-headers
Allowed and returned value	Integer value Length in bytes of the message body.

Description

- The Content-Length is optional in an HTTP request. For a **GET** or **DELETE** the length must be zero. For **POST**, if Content-Length is specified and it does not match the length of the message-line, the message is either truncated, or padded with nulls to the specified length.
- The Content-Length is always returned in the HTTP response even when there is no content, in which case the value is zero.

Content-Location: HTTP entity-header

Returns the queue or topic referenced in the request, in the standard Content-Location header in the HTTP response message.

Type	Description
HTTP header name	Content-Location
HTTP header type	Entity-header

Type	Description
Valid in HTTP request message	x-msg-require-headers
Returned value	URI in the format, <div>/msg/queue/queuename</div> or <div>/msg/topic/topicname</div>

Description

- When requested in x-msg-require-headers, the Content-Location entity-header returns the queue or topic referenced in the HTTP request.

Content-Range: HTTP entity-header

Return the range of bytes selected from an IBM MQ message in the Content-Range header in an HTTP response.

Type	Description
HTTP header name	Content-Range
HTTP header type	Entity-header
Valid in HTTP request message	x-msg-require-headers
Returned value	String Returns the lower limit, <i>m</i> and upper limit, <i>n</i> of the returned substring, and <i>length</i> of the whole message. For example, <div><i>m</i> - <i>n</i>/<i>length</i></div>

Description

-
- The Content-Range is only returned in the HTTP response when Content-Range is specified in a **GET** or **DELETE** request that contains an x-msg-range request header.
- If x-msg-range is specified on a **GET** or **DELETE** request, the range of bytes specified in the Content-Range header are returned in the response. For example, if x-msg-range: 0-60 is used in a request for a message containing 100 bytes, the content-range header holds the string 0-60/100
- An x-msg-range request also returns the content range in the x-msg-range header in the HTTP response.

Content-Type: HTTP entity-header

Set or return the class of the JMS message in an IBM MQ message according the to HTTP content-type.

Type	Description
HTTP header name	Content-Type
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers

Type	Description
Allowed or returned value	media-type For media-types that are supported see Table 663 on page 1738 .

Table 663. Mapping between x-msg-class and HTTP Content-Type

x-msg-class	HTTP Content-Type
BYTES	application/octet-stream application/xml
TEXT	text/*
MAP	application/x-www-form-urlencoded application/xml (optional)
STREAM	application/xml (optional)

Description

- On an HTTP **POST** request, specify either the Content-Type or the x-msg-class. If you specify both, they must be consistent or an HTTP Bad Request exception, Status code 400 is returned. If you omit both, the Content-Type and the x-msg-class, a Content-Type of text/* is assumed.
- The Content-Type is always set in the response to an HTTP **GET** or **DELETE** that has a message body. The Content-Type is set according to the rules in [Table 664 on page 1738](#).

Table 664. Mapping message types to x-msg-class and Content-Type

Message format	JMS Message type	x-msg-class	Content-Type
Anything except MQFMT_STRING	None	BYTES	application/octet-stream
MQFMT_STRING	None	TEXT	text/plain
MQFMT_NONE	jms_bytes	BYTES	application/octet-stream
MQFMT_NONE	jms_text	TEXT	text/plain
MQFMT_NONE	jms_map	MAP	application/xml
MQFMT_NONE	jms_stream	STREAM	application/xml

correlId: HTTP x-msg-correlId entity-header

Set or return the correlation identifier.

Type	Description
HTTP header name	x-msg-correlId
HTTP header type	Entity-header
Valid in HTTP request message	DELETE, GET, POST , x-msg-require-headers

Type	Description
Allowed values	<p>String value For example:</p> <pre>x-msg-correlId: mycorrelationid</pre> <p>Strings enclosed in quotation marks are permitted; for example:</p> <pre>x-msg-correlId: "my id"</pre> <p>Hex value A hex value prefixed with 0x : ; for example:</p> <pre>x-msg-correlId: 0x:43c1d23a</pre> <p>The hex value following 0x: is limited to 48 characters representing 24 bytes. Additional data is ignored.</p>
Default value	Not applicable

Description

- On an HTTP **POST** request, sets the correlation ID of the message created.
- On an HTTP **GET** or **DELETE** request, selects the message from the queue or topic. If no message exists with the specified correlation ID, an HTTP 504 Gateway Timeout response is returned. `x-msg-correlId` can be used with `x-msg-msgID` to select a message from a queue or topic that matches both selectors.
- Specified in `x-msg-require-headers`, sets `x-msg-coreId` in the HTTP response message to the correlation ID of a message.
- Horizontal white space is allowed after the `0x :` prefix.

Note:

- Specifying `x-msg-correlId` without a value on an HTTP **GET** or **DELETE** request; for example, `"x-msg-correlId: "`, returns the next message on the queue or topic regardless of its correlation ID.
- If you specify a selector of 24 characters or fewer, or `0x:` followed by 48 characters or fewer, IBM MQ bridge for HTTP uses an optimized selector for improved performance.
- A JMS message selector containing `JMSCorrelationID` is used when selecting messages from the queue. This selector behaves as described in [Selection behavior](#).

encoding: HTTP x-msg-encoding entity-header

Set or return the message encoding.

Type	Description
HTTP header name	<code>x-msg-encoding</code>
HTTP header type	Entity-header
Valid in HTTP request message	POST , <code>x-msg-require-headers</code>
Allowed values	<p>A comma-separated list of the following values:</p> <p>DECIMAL_NORMAL</p> <p>DECIMAL_REVERSED</p> <p>FLOAT_IEEE_NORMAL</p> <p>FLOAT_IEEE_REVERSED</p>

Type	Description
	FLOAT_S390 INTEGER_NORMAL INTEGER_REVERSED For example, <pre>x-msg-encoding: INTEGER_NORMAL,DECIMAL_NORMAL,FLOAT_IEEE_NORMAL</pre> Note: The value is case-sensitive
Default value	DECIMAL_NORMAL, FLOAT_IEEE_NORMAL, INTEGER_NORMAL

Description

- On an HTTP **POST** request, specifies the encoding of the message created.
- On an HTTP **GET** or **DELETE** request, the x-msg-encoding header is ignored.
- Specified in x-msg-require-headers, sets x-msg-encoding in the HTTP response message to the encoding property of a message.

expiry: HTTP x-msg-expiry entity-header

Set or return the message expiry duration.

Type	Description
HTTP header name	x-msg-expiry
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	UNLIMITED For example; <pre>x-msg-expiry: UNLIMITED</pre> Integer value Milliseconds before expiry. For example; <pre>x-msg-expiry: 10000</pre>
Default value	UNLIMITED

Description

- When set on an HTTP **POST** request, the request message expires in the time specified.
- On an HTTP **GET** or **DELETE** request, the x-msg-expiry header is ignored.
- Specified in x-msg-require-headers, sets x-msg-expiry in the HTTP response message to the expiry time of a message.
- UNLIMITED specifies that the message never expires.
- The expiry of a message starts from the time the message arrives on the queue, as a result network latency is ignored.
- The maximum value is limited by IBM MQ to 214748364700 milliseconds. If a value greater than this is specified then the maximum possible expiry time is assumed.

format: HTTP x-msg-format entity-header

Set or return the IBM MQ message format.

Type	Description
HTTP header name	x-msg-format
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	NONE For example, <pre>x-msg-format: NONE</pre> String value Any user-defined value of up to eight characters. For example, <pre>x-msg-format: myformat</pre>
Default value	None

Description

- When set on an HTTP **POST** request, set the request message format.
- On an HTTP **GET** or **DELETE** request, the x-msg-format header is ignored.
- Specified in x-msg-require-headers, sets x-msg-format in the HTTP response message to the format of a message.
- NONE is case sensitive, and indicates that the message format is blank.
- The value of x-msg-format is used, even if it contradicts the media-type of the HTTP request. See [Table 665 on page 1741](#).

Table 665. Mapping content-type and x-msg-class to message format		
x-msg-class	Content-type	Message format on queue/topic
BYTES	<ul style="list-style-type: none">• application/octet-stream• application/xml	IBM MQ message: MQFMT set to MQC.MQFMT_NONE
TEXT	<ul style="list-style-type: none">• text/*	IBM MQ message: MQFMT set to MQC.MQFMT_STRING
MAP	<ul style="list-style-type: none">• application/x-www-form-urlencoded• application/xml (optional)	JMSMap
STREAM	<ul style="list-style-type: none">• application/xml (optional)	JMSStream

msgId: HTTP x-msg-msgId entity-header

Set or return the message identifier.

Type	Description
HTTP header name	x-msg-msgId

Type	Description
HTTP header type	Entity-header
Valid in HTTP request message	DELETE, GET, POST , x-msg-require-headers
Allowed values	<p>String value For example,</p> <pre>x-msg-msgId: mymsgid</pre> <p>Strings enclosed in quotation marks for example, x-msg-msgId: "my id"</p> <p>Hex value A hex value prefixed with 0x : ; for example,</p> <pre>x-msg-msgId: 0x:43c1d23a</pre>
Default value	Not applicable

Description

- On an HTTP **POST** request, sets the message ID of the message created.
- On an HTTP **GET** or **DELETE** request, selects the message from the queue or topic. If no message exists with the specified message ID, an HTTP 504 Gateway Timeout response is returned. x-msg-msgId can be used with x-msg-correlID to select a message from a queue or topic that matches both selectors.
- Specified in x-msg-require-headers, returns x-msg-msgId in the HTTP response to the message ID of a message.
- Horizontal white space is allowed after the 0x : prefix.

Note: Specifying x-msg-msgId without a value on an HTTP **GET** or **DELETE** request; for example, "x-msg-msgId: ", returns the next message on the queue or topic regardless of its message ID.

A JMS message selector containing JMSMessageID is used when selecting messages from the queue. This selector behaves as described in [Selection behavior](#).

persistency: HTTP x-msg-persistence entity-header

Set or return the message persistence.

Type	Description
HTTP header name	x-msg-persistence
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	<p>NON_PERSISTENT The message does not survive system failures or queue manager restarts. For example,</p> <pre>x-msg-persistence: NON_PERSISTENT</pre> <p>PERSISTENT The message survives system failures and restarts of the queue manager.</p>

Type	Description
	<p>For example,</p> <pre>x-msg-persistence: PERSISTENT</pre> <p>AS_DESTINATION Applies to POST only. Use the default persistence of the destination as determined by the message provider.</p> <p>Note: Case sensitive</p>
Default value	NON_PERSISTENT

Description

- When set on an HTTP **POST** request, set the request message persistence.
- On an HTTP **GET** or **DELETE** request, the x-msg-persistence header is ignored.
- Specified in x-msg-require-headers, sets x-msg-persistence in the HTTP response message to the persistence of a message.

priority: HTTP x-msg-priority entity-header

Set or return the message priority.

Type	Description
HTTP header name	x-msg-priority
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	<p>LOW For example,</p> <pre>x-msg-priority: LOW</pre> <p>MEDIUM This priority is equal to an IBM MQ priority level of 4. For example,</p> <pre>x-msg-priority: MEDIUM</pre> <p>HIGH For example,</p> <pre>x-msg-priority: HIGH</pre> <p>Integer value A string representation of an integer in the range 0 through 9; for example,</p> <pre>x-msg-priority: 3</pre> <p>AS_DESTINATION Applies to POST only. Use the default priority of the destination as determined by the message provider.</p> <p>Note: Case sensitive</p>

Type	Description
Default value	MEDIUM

Description

- When set on an HTTP **POST** request, set the request message priority.
- On an HTTP **GET** or **DELETE** request, the x-msg-priority header is ignored.
- Specified in x-msg-require-headers, sets x-msg-priority in the HTTP response message to the priority of a message.

priority: HTTP x-msg-priority entity-header

Set or return the message priority.

Type	Description
HTTP header name	x-msg-priority
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	<p>LOW For example,</p> <pre>x-msg-priority: LOW</pre> <p>MEDIUM This priority is equal to an IBM MQ priority level of 4. For example,</p> <pre>x-msg-priority: MEDIUM</pre> <p>HIGH For example,</p> <pre>x-msg-priority: HIGH</pre> <p>Integer value A string representation of an integer in the range 0 through 9; for example,</p> <pre>x-msg-priority: 3</pre> <p>AS_DESTINATION Applies to POST only. Use the default priority of the destination as determined by the message provider.</p> <p>Note: Case sensitive</p>
Default value	MEDIUM

Description

- When set on an HTTP **POST** request, set the request message priority.
- On an HTTP **GET** or **DELETE** request, the x-msg-priority header is ignored.
- Specified in x-msg-require-headers, sets x-msg-priority in the HTTP response message to the priority of a message.

replyTo: HTTP x-msg-replyTo entity-header

Set or return the message reply-to queue and queue manager name.

Type	Description
HTTP header name	x-msg-replyTo
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	URI A point-to-point URI; for example, <pre>x-msg-replyTo: /msg/queue/myReplyQueue x-msg-replyTo: /msg/queue/myReplyQueue@myReplyQueueManager</pre> Note: Case sensitive
Default value	MEDIUM

Description

- When set on an HTTP **POST** request, set the request message replyTo destination.
- On an HTTP **GET** or **DELETE** request, the x-msg-replyTo header is ignored.
- Specified in x-msg-require-headers, sets x-msg-replyTo in the HTTP response message to the reply-to queue and queue manager name of a message.

Note: The URI in the HTTP response can include the name of the queue manager to which the IBM MQ bridge for HTTP is connected.

Server: HTTP response-header

Returns information about the server and protocol the client is connected to.

Type	Description
HTTP header name	Server
HTTP header type	Response-header
Valid in HTTP request message	x-msg-require-headers
Returned value	<pre>WMQ-HTTP/1.1 JEE-Bridge/1.1</pre> or <pre>Server: Product-token WMQ-HTTP/1.1 JEE-Bridge/1.1</pre>

Description

- If IBM MQ Bridge for HTTP is deployed to an application server, the IBM MQ bridge for HTTP details is appended to the server response header. For example, the IBM MQ bridge for HTTP deployed to WebSphere Application Server Community Edition, called Apache-Coyote, gives the response:

```
Server: Apache-Coyote/1.1 WMQ-HTTP/1.1 JEE-Bridge/1.1
```

require-headers: HTTP x-msg-require-headers request-header

Set which headers to return in the HTTP response message.

Type	Description
HTTP header name	x-msg-require-headers
HTTP header type	Request-header
Valid in HTTP request message	POST, GET, DELETE
Allowed values	<p>A comma-separated list of the entity header names:</p> <p>ALL</p> <p>ALL-USR</p> <p>class</p> <p>content-location</p> <p>correlId</p> <p>encoding</p> <p>expiry</p> <p>format</p> <p>msgId</p> <p>NO_require-headers</p> <p>persistence</p> <p>priority</p> <p>replyTo</p> <p>server</p> <p>timestamp</p> <p>usr- <i>property name</i></p> <p>For example,</p> <pre>x-msg-require-headers: msgId</pre> <p>or,</p> <pre>x-msg-require-headers: expiry,correlId,timestamp</pre> <p>To request a specific property:</p> <pre>x-msg-require-headers: usr-myCustomProperty</pre> <p>To request all properties:</p> <pre>x-msg-require-headers: ALL-USR, ALL</pre>
Default value	NO_require-headers

Description

- The value of x-msg-require-headers is not case-sensitive, except in the cases of the ALL, NO_require-headers, and ALL-USR constants, and the *property-name* variable.

timestamp: HTTP x-msg-timestamp entity-header

Return the message time stamp.

Type	Description
HTTP header name	x-msg-timestamp
HTTP header type	Entity-header
Valid in HTTP request message	x-msg-require-headers
Returned value	HTTP-date A date in the format; day, date month year time time-zone; for example, <pre>Sun, 06 Nov 1994 08:49:37 GMT</pre> Defined by RFC 822, and updated in RFC 1123.
Default value	Not applicable

Description

- On an HTTP **POST**, **GET** or **DELETE** request, the x-msg-timestamp header is ignored.
- Specified in x-msg-require-headers, sets x-msg-timestamp in the HTTP response message to the timestamp of a message.

usr: HTTP x-msg-usr entity-header

Set or return the user properties.

Type	Description
HTTP header name	x-msg-usr
HTTP header type	Entity-header
Valid in HTTP request message	POST , x-msg-require-headers
Allowed values	See “Syntax” on page 1748 ; for example, <pre>x-msg-usr: myProp1;5;i1, x-msg-usr: myProp2;"My String";string</pre>
Default value	Not applicable

Description

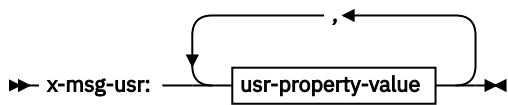
- When set on an HTTP **POST** request, set the request message user properties.
- On an HTTP **GET** or **DELETE** request, the x-msg-usr header is ignored.
- Specified in x-msg-require-headers, sets x-msg-usr in the HTTP response message to user properties of a message.
- Multiple properties can be set on a message. Specify multiple comma-separated properties in a single x-msg-usr header, or by using two or more separate instances of the x-msg-usr header.
- You can request a specific property to be returned in the response to a **GET** or **DELETE** request. Specify the name of the property in the x-msg-require-headers header of the request, using the prefix **usr-**. For example,

```
x-msg-require-headers: usr-myProp1
```

- To request that all user properties are returned in a response, use the ALL-USR constant. For example,

```
x-msg-require-headers: ALL-USR
```

Syntax



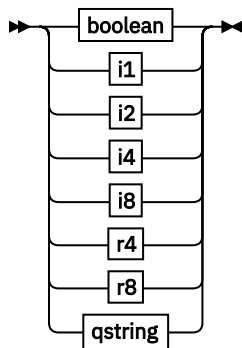
usr-property-value



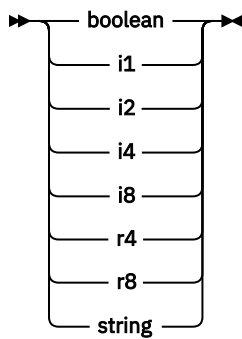
property-name



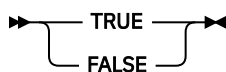
usr-value



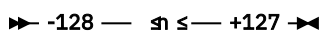
usr-type



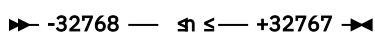
boolean



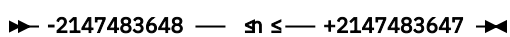
i1



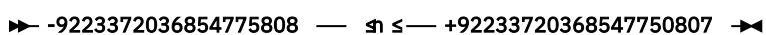
i2



i4



i8



r4

➤ -1.4E-45 — \leq — +3.4028235E38 ➤

r8

➤ -4.9E-324 — \leq — +1.7976931348623157E308 ➤

qstring

➤ " — *string* — " ➤

wait: HTTP x-msg-wait request-header

Set the period of time to wait for a message to arrive, before returning an HTTP 504 Gateway Timeout response message.

Type	Description
HTTP header name	x-msg-wait
HTTP header type	Request-header
Valid in HTTP request message	GET, DELETE
Allowed value	NO_WAIT For example, <pre>x-msg-wait: NO_WAIT</pre> Integer value The number of milliseconds that the IBM MQ bridge for HTTP waits for a message to arrive; for example, <pre>x-msg-wait: 8</pre>
Default value	NO_WAIT

Description

- On an HTTP **POST** request, the x-msg-wait header is ignored.
- On an HTTP **GET** or **DELETE** request, x-msg-wait specifies time to wait for a message to arrive before returning an HTTP 504 Gateway Timeout response.
- NO_WAIT is case-sensitive.
- The default maximum wait time is 35000. You can change the default by setting the maximum_wait_time parameter of the servlet. See the [Installing, configuring, and verifying IBM MQ bridge for HTTP](#) section for more information.
- If you set a value greater than maximum_wait_time, maximum_wait_time is used instead.

HTTP return codes

List of return codes from the IBM MQ bridge for HTTP

The IBM MQ bridge for HTTP returns four types of error:

Servlet errors

MQHTTP0001 and MQHTTP0002 are servlet errors. They are logged, but not returned to the HTTP client.

Successful operations

An HTTP status code in the range 200 - 299 indicates a successful operation.

Client errors

An HTTP status code in the range 400 - 499 indicates a client error. IBM MQ Bridge for HTTP return codes in the range MQHTTP40001 - MQHTTP49999 correspond to client errors.

Server errors

An HTTP status code in the range 500 - 599 indicates a client error. IBM MQ Bridge for HTTP return codes in the range MQHTTP50001 - MQHTTP59999 correspond to server errors.

If a server error occurs, a complete stack trace is output to the application server error log. The stack trace is also returned to the HTTP client in the HTTP response. Handle the stack trace in the client application or refer it to the application server administrator to resolve the problem.

If the stack trace contains resource adapter errors, refer to the documentation for your resource adapter.

Alphabetic list of return codes

HTTP 200: OK

This class of status code indicates that the request was successfully received, understood and accepted.

HTTP status code

200 OK

HTTP 204: No content

Sent following a successful HTTP **GET** or **DELETE** and x-msg-range: 0 was sent in the request.

HTTP status code

204 No Content

MQHTTP0001: No connection factory specified in the Servlet context

Servlet error

Explanation

Servlet error

HTTP status code

None

Programmer response

Where these errors are logged is specific to your application server. Refer to the documentation for your application server.

MQHTTP0002: Could not get connection manager for *queueOrTopic* using the JNDI name of *jndiNameTried*

Servlet error

Explanation

Servlet error

HTTP status code

None

Programmer response

Where these errors are logged is specific to your application server. Refer to the documentation for your application server.

MQHTTP40001: Reserved

Reserved

HTTP status code

400 Bad Request

MQHTTP40002: URI is not valid for IBM MQ transport for HTTP

The URI specified in the HTTP request is not valid.

Explanation

The URI specified in the HTTP request is not valid.

HTTP status code

400 Bad Request

Programmer response

Confirm that the format and syntax of the specified URI are correct.

MQHTTP40003: URI is not valid. @qmgr is only valid on POST

The @qmgr URI option has been specified in a URI for an HTTP request that is not a **POST** request.

Explanation

The @qmgr URI option has been specified in a URI for an HTTP request that is not a **POST** request.

HTTP status code

400 Bad Request

Programmer response

If you are attempting to put a message by using the **POST** verb, change the HTTP request to a **POST** request. If you are attempting to get a message by using the **DELETE** or **GET** verbs, remove @qmgr from the URI.

MQHTTP40004: Invalid Content-Type specified

The Content-Type header field specified on a **POST** request is not compatible with the x-msg-class header value.

Explanation

The Content-Type header field specified on a **POST** request is not compatible with the x-msg-class header value.

HTTP status code

400 Bad Request

Programmer response

Change the Content-Type header field to one that is supported. The Content-Type header must be compatible with the specified x-msg-class header field.

MQHTTP40005: Bad message header value

A supported header field has been specified with a value that is not valid for the specified request.

Explanation

A supported header field has been specified with a value that is not valid for the specified request.

HTTP status code

400 Bad Request

Programmer response

Change the value specified for the given header field to a value which is valid. Check the case of the value specified, as some header fields have case-sensitive values.

MQHTTP40006: *Header_name* is not a valid request header

A header that is valid only in an HTTP response message has been specified in an HTTP request message.

Explanation

A header which is valid only in an HTTP response message has been specified in an HTTP request message.

HTTP status code

400 Bad Request

Programmer response

Remove any headers from the HTTP request which are only valid in an HTTP response; for example, x-msg-timestamp.

MQHTTP40007: *Header_name* is only valid on ...

A header has been specified in an HTTP request, but the header field is not valid for the given request verb.

Explanation

A header has been specified in an HTTP request, but the header field is not valid for the given request verb.

HTTP status code

400 Bad Request

Programmer response

Remove any headers from the HTTP request which are not valid for the given request verb. For example, x-msg-encoding is valid for HTTP **POST** requests, but not valid for HTTP **GET** or HTTP **DELETE** requests.

MQHTTP40008: *Header_name* maximum length is ...

The maximum length for the given header field has been exceeded.

Explanation

The maximum length for the given header field has been exceeded.

HTTP status code

400 Bad Request

Programmer response

Change the value of the header field to a value which is within the range permitted for the header field.

MQHTTP40009: Header field *header_field* is not valid for ...

A header field specified in an HTTP request is not supported by the messaging provider to which the IBM MQ bridge for HTTP is connected.

Explanation

A header field specified in an HTTP request is not supported by the messaging provider to which the IBM MQ bridge for HTTP is connected. The error occurs if a messaging provider is used that cannot support all the features of the IBM MQ bridge for HTTP.

HTTP status code

400 Bad Request

Programmer response

Remove the unsupported header from the HTTP request.

MQHTTP40010: Message with Content-Type *content_type* could not be parsed

The content of the HTTP request is not compatible with the Content-Type of the request.

Explanation

The content of the HTTP request is not compatible with the Content-Type of the request. A common cause is badly formed application/x-www-form-urlencoded or application/xml data.

HTTP status code

400 Bad Request

Programmer response

Correct the content of the HTTP request so that it is in the correct format for the Content-Type of the request.

MQHTTP40301: You are forbidden from accessing ...

The IBM MQ bridge for HTTP has been unable to authenticate itself for the specified destination.

Explanation

The IBM MQ bridge for HTTP has been unable to authenticate itself for the specified destination.

HTTP status code

403 Forbidden

Programmer response

Change the authentication properties of the destination so that the IBM MQ Bridge for HTTP is authorized to connect to it. Alternatively, specify a destination to which the IBM MQ bridge for HTTP is authorized to connect.

MQHTTP40302: You are forbidden from ...

The IBM MQ bridge for HTTP has been unable to connect to the queue manager.

Explanation

The IBM MQ bridge for HTTP has been unable to connect to the queue manager. The IBM MQ bridge for HTTP security configuration is incorrect.

HTTP status code

403 Forbidden

Programmer response

Change the authentication configuration of the queue manager so that the IBM MQ Bridge for HTTP is authorized to connect to it. Alternatively, configure the IBM MQ bridge for HTTP to connect to a queue manager to which it is authorized to connect.

MQHTTP40401: The destination *destination_name* could not be found

The destination specified in the HTTP request URI cannot be found by the IBM MQ bridge for HTTP.

Explanation

The destination specified in the HTTP request URI cannot be found by the IBM MQ bridge for HTTP.

HTTP status code

404 Not found

Programmer response

Check that the destination specified in the HTTP request URI exists, or specify an alternative destination.

MQHTTP40501: Method *method_name* not allowed

The method specified in the HTTP request is not supported by the IBM MQ bridge for HTTP.

Explanation

The method specified in the HTTP request is not supported by the IBM MQ bridge for HTTP.

HTTP status code

405 Method not allowed

Programmer response

Change the method specified in the HTTP request to one which is supported by the IBM MQ bridge for HTTP.

MQHTTP41301: The message being posted was too large for the destination

The destination specified in the HTTP POST request URI cannot accept messages that are as long as the message specified in the HTTP request.

Explanation

The destination specified in the HTTP POST request URI cannot accept messages that are as long as the message specified in the HTTP request.

HTTP status code

413 Request entity too large

Programmer response

Reduce the size of the message specified in the HTTP request. Alternatively, specify a destination which can support messages of the wanted length.

MQHTTP41501: The media type character set is unsupported

The character set specified in the Content-Type header field is not supported by the IBM MQ bridge for HTTP.

Explanation

The character set specified in the Content-Type header field is not supported by the IBM MQ bridge for HTTP.

HTTP status code

415 Unsupported media type

Programmer response

Change the character set of the Content-Type header field to one that is supported by the IBM MQ bridge for HTTP.

MQHTTP41502: Media-type *media-type* is not supported ...

The media-type specified in the HTTP request is not supported by the IBM MQ bridge for HTTP for the specified HTTP verb.

Explanation

The media-type specified in the HTTP request is not supported by the IBM MQ bridge for HTTP for the specified HTTP verb.

HTTP status code

415 Unsupported media type

Programmer response

Change the media-type specified in the HTTP request to one that is supported by the IBM MQ Bridge for HTTP for the specified HTTP verb.

MQHTTP41503: Media-type *media-type* is not supported ...

The media-type specified in the HTTP request is not supported by the IBM MQ bridge for HTTP for the specified x-msg-class header field.

Explanation

The media-type specified in the HTTP request is not supported by the IBM MQ bridge for HTTP for the specified x-msg-class header field.

HTTP status code

415 Unsupported media type

Programmer response

Change the media-type specified in the HTTP request to one that is supported by the IBM MQ Bridge for HTTP for the specified x-msg-class header field.

MQHTTP41701: The HTTP header Expect is not supported

The IBM MQ bridge for HTTP does not support the Expect header field.

Explanation

The Expect header has been specified in an HTTP request. The IBM MQ bridge for HTTP does not support the Expect header field.

HTTP status code

417 Expectation failed

Programmer response

Remove the Expect header from the HTTP request.

MQHTTP50001: There has been an unexpected problem ...

An error has occurred in the IBM MQ bridge for HTTP.

Explanation

An error has occurred in the IBM MQ bridge for HTTP.

HTTP status code

500 Internal server error

Programmer response

Contact the system administrator of the IBM MQ Bridge for HTTP.

MQHTTP50201: An error has occurred between the IBM MQ bridge for HTTP and the queue manager

An error has occurred between the IBM MQ bridge for HTTP and the queue manager

Explanation

An error has occurred between the IBM MQ bridge for HTTP and the queue manager

HTTP status code

502 Bad Gateway

Programmer response

Contact the system administrator of the IBM MQ Bridge for HTTP.

MQHTTP50401: Message retrieval timed out

No message matching the specified request parameters in an HTTP **GET** or HTTP **DELETE** was returned in the timeout period.

Explanation

No message matching the specified request parameters in an HTTP **GET** or HTTP **DELETE** was returned in the timeout period. The return code indicates that no suitable message was available at any time during the life of the HTTP request.

HTTP status code

504 Gateway timeout

Programmer response

If a message was expected, check the header fields of the HTTP request such as `x-msg-correlId` and `x-msg-msgid`. Check that the destination specified in the HTTP request URI is correct. Try extending the wait time of the HTTP request using the `x-msg-wait` header field.

MQHTTP50501: HTTP 1.1 and upwards ...

The HTTP protocol used in the HTTP request is not supported by the IBM MQ bridge for HTTP.

Explanation

The HTTP protocol used in the HTTP request is not supported by the IBM MQ bridge for HTTP.

HTTP status code

505 HTTP version not supported

Programmer response

Change the HTTP request to use HTTP protocol V1.1 or higher.

Message types and message mappings for WebSphere Bridge for HTTP

IBM MQ bridge for HTTP supports four message classes, TEXT, BYTES, STREAM and MAP. The message classes are mapped to JMS message types and HTTP Content-Type.

HTTP POST

The message type that arrives at the destination depends on the value of the `x-msg-class` header or the Content-Type of the HTTP request. [Table 666 on page 1757](#) shows the HTTP Content-Type type that corresponds to each `x-msg-class`. Either field can be used to set the message type and message format. If both fields are set, and are set inconsistently, then a `Bad Request` exception is returned (HTTP 400, MQHTTP20004).

Table 666. Mapping between <i>x-msg-class</i> and HTTP Content-Type	
x-msg-class	HTTP Content-Type
BYTES	application/octet-stream application/xml
TEXT	text/*
MAP	application/x-www-form-urlencoded application/xml (optional)
STREAM	application/xml (optional)

If the JMS message type is set in the MQRFH2 header, it is mapped according to [Table 667 on page 1757](#).

Table 667. Mapping between <i>x-msg-class</i> and JMS message types.	
x-msg-class	JMS message type
BYTES	jms_bytes
TEXT	jms_text
MAP	jms_map

Table 667. Mapping between <i>x-msg-class</i> and JMS message types. (continued)	
x-msg-class	JMS message type
STREAM	jms_stream

The JMS message type is always set for a message class of MAP or STREAM. It is not always set for a message class of BYTES or TEXT. If a MQRFH2 is to be created for the request, the JMS message type is always set. Otherwise, if no MQRFH2 is created, no JMS message type is set. An MQRFH2 is created if user properties are set in the request, using the `x-msg-user` header.

If the JMS message type is set, then the message format is set to MQFMT_NONE, see [Table 669 on page 1758](#):

Table 668. Mapping between <i>x-msg-class</i> and IBM MQ message format		
x-msg-class	Message format with MQRFH2 present in message	Message format with no MQRFH2 present in message
BYTES	MQFMT_NONE	MQFMT_NONE
TEXT	MQFMT_NONE	MQFMT_STRING
MAP	MQFMT_NONE	Not possible
STREAM	MQFMT_NONE	Not possible

HTTP GET or DELETE

The message type or format retrieved determines the value of the `x-msg-class` header and the Content-Type of the HTTP response. The `x-msg-class` header is returned only if requested in a `x-msg-headers` request.

[Table 669 on page 1758](#) describes the mappings between `x-msg-class` and Content-Type, and the message type retrieved from the queue or topic.

Table 669. Mapping message types to <i>x-msg-class</i> and Content-Type			
Message format	JMS Message type	x-msg-class	Content-Type
Anything except MQFMT_STRING	None	BYTES	application/octet-stream
MQFMT_STRING	None	TEXT	text/plain
MQFMT_NONE	jms_bytes	BYTES	application/octet-stream
MQFMT_NONE	jms_text	TEXT	text/plain
MQFMT_NONE	jms_map	MAP	application/xml
MQFMT_NONE	jms_stream	STREAM	application/xml

MAP and STREAM message class serialization

MAP and STREAM message classes are serialized back to the client in the HTTP response in the same way as a message is serialized to a queue.

For MAP, the XML name, type, and value triplets are encoded as:

```
<map>
  <elt name="elementname1" dt="datatype1">value1</elt>
  <elt name="elementname2" dt="datatype2">value2</elt>
```

```
</map>
```

STREAM is like MAP, but it does not have element names:

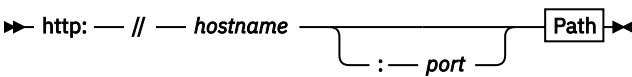
```
<stream>
  <elt dt="datatype1">value1</elt>
  <elt dt="datatype2">value2</elt>
  ...
</stream>
```

Note: datatype is one of the data types defined for defining user-defined properties and listed in [“usr: HTTP x-msg-usr entity-header” on page 1747](#). The attribute dt="string" is omitted for string elements because the default data type is string.

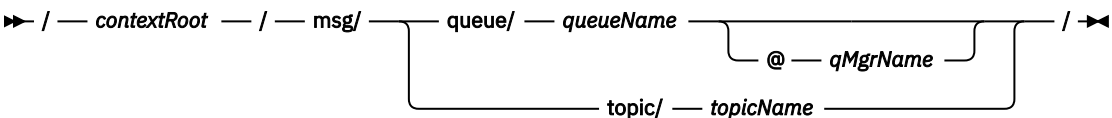
URI Format

URIs intercepted by IBM MQ bridge for HTTP.

Syntax



Path



Note:

1. If a question mark (?) is used it must be substituted with %3f. For example, orange?topic should be specified as orange%3ftopic.
2. @qMgrName is only valid on an HTTP **POST**

Description

Deploy the IBM MQ bridge for HTTP servlet to your JEE application server with a context root of *contextRoot*. Requests to

```
http://hostname: port/context_root/msg/queue/queueName @ qMgrName
```

and

```
http://hostname: port/context_root/msg/topic/topicString
```

are intercepted by IBM MQ bridge for HTTP.

The IBM MQ .NET classes and interfaces

IBM MQ .NET classes and interfaces are listed alphabetically. The properties, methods and constructors are described.

MQAsyncStatus.NET class

Use MQAsyncStatus to inquire on the status of previous MQI activity; for example inquiring on the success of previous asynchronous put operations. MQAsyncStatus encapsulates features of the MQSTS data structure.

Class

```
System.Object
├── IBM.WMQ.MQBase
│   └── IBM.WMQ.MQBaseObject
│       └── IBM.WMQ.MQAsyncStatus
```

`public class IBM.WMQ.MQAsyncStatus extends IBM.WMQ.MQBaseObject;`

- [“Properties” on page 1760](#)
- [“Constructors” on page 1761](#)

Properties

Test for MQException being thrown when getting properties.

public static int CompCode {get;}

The completion code from the first error or warning.

public static int Reason {get;}

The reason code from the first error or warning.

public static int PutSuccessCount {get;}

The number of successful asynchronous MQI put calls.

public static int PutWarningCount {get;}

The number of asynchronous MQI put calls that succeeded with a warning.

public static int PutFailureCount {get;}

The number of failed asynchronous MQI put calls.

public static int ObjectType {get;}

The object type for the first error. The following values are possible:

- MQC.MQOT_ALIAS_Q
- MQC.MQOT_LOCAL_Q
- MQC.MQOT_MODEL_Q
- MQC.MQOT_Q
- MQC.MQOT_REMOTE_Q
- MQC.MQOT_TOPIC
- 0, meaning that no object is returned

public static string ObjectName {get;}

The object name.

public static string ObjectQMgrName {get;}

The object queue manager name.

public static string ResolvedObjectName {get;}

The resolved object name.

public static string ResolvedObjectQMgrName {get;}

The resolved object queue manager name.

Constructors

public MQAsyncStatus() throws MQException;

Constructor method, constructs an object with fields initialized to zero or blank as appropriate.

MQAuthenticationInformationRecord.NET class

Use `MQAuthenticationInformationRecord` to specify information about an authenticator that is to be used in an IBM MQ SSL client connection. `MQAuthenticationInformationRecord` encapsulates an authentication information record, MQAIR.

Class

```
System.Object
└─ IBM.WMQ.MQAuthenticationInformationRecord
```

public class IBM.WMQ.MQAuthenticationInformationRecord extends System.Object;

- [“Properties” on page 1761](#)
- [“Constructors” on page 1762](#)

Properties

Test for `MQException` being thrown when getting properties.

public long Version {get; set;}

Structure version number.

public long AuthInfoType {get; set;}

The type of authentication information. This attribute must be set to one of the following values:

- OCSP - Certificate revocation status checking is done using OCSP.
- CRLLDAP - Certificate revocation status checking is done using Certificate Revocation Lists on LDAP servers.

public string AuthInfoConnName {get; set;}

The DNS name or IP address of the host on which the LDAP server is running, with an optional port number. This keyword is required.

public string LDAPPassword {get; set;}

The password associated with the distinguished name of the user who is accessing the LDAP server. This property applies only when **AuthInfoType** is set to CRLLDAP.

```
public string LDAPUserName {get; set;}
```

The distinguished name of the user who is accessing the LDAP server. When you set this property, LDAPUserNameLength and LDAPUserNamePtr are automatically set correctly. This property applies only when AuthInfoType is set to CRLLDAP.

```
public string OCSPResponderURL {get; set;}
```

The URL at which the OCSP responder can be contacted. This property applies only when AuthInfoType is set to OCSP

This field is case sensitive. It must start with the string http:// in lowercase. The rest of the URL might be case sensitive, depending on the OCSP server implementation.

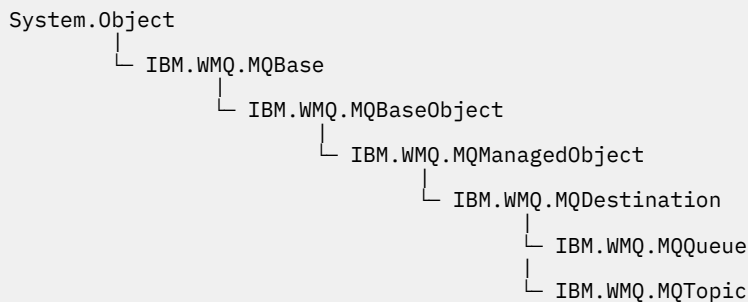
Constructors

```
MQAuthenticationInformationRecord();
```

MQDestination.NET class

Use MQDestination to access methods that are common to MQQueue and MQTopic. MQDestination is an abstract base class and cannot be instantiated.

Class



```
public class IBM.WMQ.MQDestination extends IBM.WMQ.MQManagedObject;
```

- [“Properties” on page 1762](#)
- [“Methods” on page 1763](#)
- [“Constructors” on page 1764](#)

Properties

Test for MQException being thrown when getting properties.

```
public DateTime CreationDateTime {get;}
```

The date and time that the queue or topic was created. Originally contained within MQQueue, this property has been moved into the base MQDestination class.

There is no default value.

```
public int DestinationType {get;}
```

Integer value describing the type of destination being used. Initialized from the sub classes constructor, MQQueue or MQTopic, this value can take one of these values:

- MQOT_Q
- MQOT_TOPIC

There is no default value.

Methods

```
public void Get(MQMessage message);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions, int  
MaxMsgSize);
```

Throws MQException.

Gets a message from a queue if the destination is an MQQueue object or from a topic if the destination is an MQTopic object, using a default instance of MQGetMessageOptions to do the get.

If the get fails, the MQMessage object is unchanged. If it succeeds, the message descriptor and message data portions of the MQMessage are replaced with the message descriptor and message data from the incoming message.

All calls to IBM MQ from a particular MQQueueManager are synchronous. Therefore, if you perform a get with wait, all other threads using the same MQQueueManager are blocked from making further IBM MQ calls until the Get call is accomplished. If you need multiple threads to access IBM MQ simultaneously, each thread must create its own MQQueueManager object.

message

Contains the message descriptor and the returned message data. Some of the fields in the message descriptor are input parameters. It is important to ensure that the MessageId and CorrelationId input parameters are set as required.

A reconnectable client returns the reason code MQRC_BACKED_OUT after successful reconnection, for messages received under MQGM_SYNCPOINT.

getMessageOptions

Options controlling the action of the get.

Using option MQC.MQGM_CONVERT might result in an exception with reason code MQC.MQRC_CONVERTED_STRING_TOO_BIG when converting from single-byte character codes to double byte codes. In this case, the message is copied into the buffer without conversion.

If *getMessageOptions* is not specified, the message option used is MQGM_NOWAIT.

If you use the MQGM_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

MaxMsgSize

The largest message this message object is to receive. If the message on the queue is larger than this size, one of two things occurs:

- If the MQGM_ACCEPT_TRUNCATED_MSG flag is set in the MQGetMessageOptions object, the message is filled with as much of the message data as possible. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_ACCEPTED reason code.
- If the MQGM_ACCEPT_TRUNCATED_MSG flag is not set, the message remains on the queue. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_FAILED reason code.

If *MaxMsgSize* is not specified, the whole message is retrieved.

```
public void Put(MQMessage message);  
public void Put(MQMessage message, MQPutMessageOptions putMessageOptions);
```

Throws MQException.

Puts a message to a queue if the destination is an MQQueue object or publishes a message to a topic if the destination is an MQTopic object.

Modifications to the MQMessage object after the Put call has been accomplished do not affect the actual message on the IBM MQ queue or publication topic.

Put updates the MessageId and CorrelationId properties of the MQMessage object and does not clear message data. Further Put or Get calls refer to the updated information in the MQMessage object. For example, in the following code snippet, the first message contains a and the second ab.

```
msg.WriteString("a");
q.Put(msg,pmo);
msg.WriteString("b");
q.Put(msg,pmo);
```

message

An MQMessage object containing the message descriptor data, and message to be sent. The message descriptor can be altered as a consequence of this method. The values in the message descriptor immediately after the completion of this method are the values that were put to the queue or published to the topic.

The following reason codes are returned to a reconnectable client:

- MQRC_CALL_INTERRUPTED if the connection is broken while running a Put call on a persistent message and the reconnection is successful.
- MQRC_NONE if the connection is successful while running a Put call on a non-persistent message (see [Application Recovery](#)).

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of MQPutMessageOptions is used.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

Note: For simplicity and performance, if you want to put a single message to a queue, use MQQueueManager.Put object. You should have an MQQueue object for this.

Constructors

MQDestination is an abstract base class and cannot be instantiated. Access destinations using MQQueue and MQTopic constructors, or using MQQueueManager.AccessQueue and MQQueueManager.AccessTopic methods.

MQEnvironment.NET class

Use MQEnvironment to control how the MQQueueManager constructor is called and to select an IBM MQ MQI client connection. The MQEnvironment class contains properties that control the behavior of the IBM MQ.

Class

```
System.Object
└─ IBM.WMQ.MQEnvironment
```

```
public class IBM.WMQ.MQEnvironment extends System.Object;
```

- [“Properties - client only” on page 1764](#)
- [“Properties” on page 1765](#)
- [“Constructors” on page 1766](#)

Properties - client only

Test for MQException being thrown when getting properties.

public static int CertificateValPolicy {get; set;}

Set which SSL/TLS certificate validation policy is used to validate digital certificates received from remote partner systems. Valid values are:

- MQC.CERTIFICATE_VALIDATION_POLICY_ANY
- MQC.CERTIFICATE_VALIDATION_POLICY_RFC5280

public static ArrayList EncryptionPolicySuiteB {get; set;}

Set the level of Suite B compliant cryptography. Valid values are:

- MQC.MQ_SUITE_B_NONE - This is the default value.
- MQC.MQ_SUITE_B_128_BIT
- MQC.MQ_SUITE_B_192_BIT

public static string Channel {get; set;}

The name of the channel to connect to the target queue manager. You *must* set the channel property before instantiating an MQQueueManager instance in client mode.

public static int FipsRequired {get; set;}

Specify MQC.MQSSL_FIPS_YES to use only FIPS-certified algorithms if cryptography is carried out in IBM MQ. The default is MQC.MQSSL_FIPS_NO.

If cryptographic hardware is configured, the cryptographic modules used are those provided by the hardware product. Depending on the hardware in use, these might not be FIPS-certified to a particular level.

public static string Hostname {get; set;}

The TCP/IP host name of the computer on which the IBM MQ server resides. If the host name is not set, and no overriding properties are set, server bindings mode is used to connect to the local queue manager.

public static int Port {get; set;}

The port to connect to. This is the port on which the IBM MQ server is listening for incoming connection requests. The default value is 1414.

public static string SSLCipherSpec {get; set;}

Set SSLCipherSpec to the value of the CipherSpec set on the SVRCONN channel to enable SSL for the connection. The default is Null, and SSL is not enabled for the connection.

public static string sslPeerName {get; set;}

A distinguished name pattern. If sslCipherSpec is set, this variable can be used to ensure that the correct queue manager is used. If set to null (default), the DN of the queue manager is not performed. sslPeerName is ignored if sslCipherSpec is null.

Properties

Test for MQException being thrown when getting properties.

public static ArrayList HdrCompList {get; set;}

Header Data Compression List

public static int KeyResetCount {get; set;}

Indicates the number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated.

public static ArrayList MQAIRArray {get; set;}

An array of MQAuthenticationInformationRecord objects.

public static ArrayList MsgCompList {get; set;}

Message Data Compression List

public static string Password {get; set;}

The password to be authenticated. The password referenced from the MQCSP structure gets populated by setting this Password property.

public static string ReceiveExit {get; set;}

A receive exit allows you to examine and alter data received from a queue manager. It is normally used with a corresponding send exit at the queue manager. If ReceiveExit is set to null, no receive exit is called.

public static string ReceiveUserData {get; set;}

The user data associated with a receive exit. Limited to 32 characters.

public static string SecurityExit {get; set;}

A security exit allows you to customize the security flows that occur when an attempt is made to connect to a queue manager. If SecurityExit is set to null, no security exit is called.

public static string SecurityUserData {get; set;}

The user data associated with a security exit. Limited to 32 characters.

public static string SendExit {get; set;}

A send exit allows you to examine or alter the data sent to a queue manager. It is normally used with a corresponding receive exit at the queue manager. If SendExit is set to null, no send exit is called.

public static string SendUserData {get; set;}

The user data associated with a send exit. Limited to 32 characters.

public static string SharingConversations {get; set;}

The SharingConversations field is used on connections from .NET applications, when these applications are not using a client channel definition table (CCDT).

SharingConversations determines the maximum number of conversations that can be shared on a socket associated with this connection.

A value of 0 means that the channel operates as it did before IBM WebSphere MQ 7.0, with regard to conversation sharing, read ahead, and heartbeat.

The field is passed in the hash table of properties as a SHARING_CONVERSATIONS_PROPERTY, when instantiating an IBM MQ queue manager.

If you do not specify SharingConversations, a default value of 10 is used.

public static string SSLCryptoHardware {get; set;}

Sets the name of the parameter string required to configure the cryptographic hardware present on the system. SSLCryptoHardware is ignored if sslCipherSpec is null.

public static string SSLKeyRepository {get; set;}

Set the fully qualified file name of the key repository.

If SSLKeyRepository is set to null (default), the certificate MQSSLKEYR environment variable is used to locate the key repository. SSLCryptoHardware is ignored if sslCipherSpec is null.

Note: The .kdb extension is a mandatory part of the file name, but is not included as part of the value of the parameter. The directory you specify must exist. IBM MQ creates the file the first time it accesses the new key repository, unless the file already exists.

public static string UserId {get; set;}

The user ID to be authenticated. The user ID referenced from the MQCSP structure gets populated by setting UserId. Authenticate UserId using an API or Security exit.

Constructors

public MQEnvironment()

MQException.NET class

Use MQException to find out the completion and reason code of a failed IBM MQ function. An MQException is thrown whenever an IBM MQ error occurs.

Class

```
System.Object
├── System.Exception
│   └── System.ApplicationException
│       └── IBM.WMQ.MQException
```

```
public class IBM.WMQ.MQException extends System.ApplicationException;
```

- [“Properties” on page 1767](#)
- [“Constructors” on page 1767](#)

Properties

public int CompletionCode {get; set;}

The IBM MQ completion code associated with the error. The possible values are:

- MQException.MQCC_OK
- MQException.MQCC_WARNING
- MQException.MQCC_FAILED

public int ReasonCode {get; set;}

IBM MQ reason code describing the error.

Constructors

public MQException(int *completionCode*, int *reasonCode*)

completionCode

The IBM MQ completion code.

reasonCode

The IBM MQ completion code.

MQGetMessageOptions.NET class

Use MQGetMessageOptions to specify how messages are retrieved. It modifies the behavior of MQDestination.Get.

Class

```
System.Object
├── IBM.WMQ.MQBase
│   └── IBM.WMQ.MQBaseObject
│       └── IBM.WMQ.MQGetMessageOptions
```

```
public class IBM.WMQ.MQGetMessageOptions extends IBM.WMQ.MQBaseObject;
```

- [“Properties” on page 1768](#)
- [“Constructors” on page 1770](#)

Properties

Note: The behavior of some of the options available in this class depends on the environment in which they are used. These elements are marked with an asterisk *.

Test for MQException being thrown when getting properties.

public int GroupStatus {get;}*

GroupStatus indicates whether the retrieved message is in a group and if it is the last in the group. Possible values are:

MQC.MQGS_LAST_MSG_IN_GROUP

Message is the last or only message in the group.

MQC.MQGS_MSG_IN_GROUP

Message is in a group, but is not the last in the group.

MQC.MQGS_NOT_IN_GROUP

Message is not in a group.

public int MatchOptions {get; set;}*

MatchOptions determines how a message is selected. The following match options can be set:

MQC.MQMO_MATCH_CORREL_ID

Correlation ID to be matched.

MQC.MQMO_MATCH_GROUP_ID

Group ID to be matched.

MQC.MQMO_MATCH_MSG_ID

Message ID to be matched.

MQC.MQMO_MATCH_MSG_SEQ_NUMBER

Match message sequence number.

MQC.MQMO_NONE

No matching required.

public int Options {get; set;}

Options control the action of MQQueue.get. Any of the following values can be specified. If more than one option is required, the values can be added, or combined using the bitwise OR operator.

MQC.MQGMO_ACCEPT_TRUNCATED_MSG

Allow truncation of message data.

MQC.MQGMO_ALL_MSGS_AVAILABLE*

Retrieve messages from a group only when all the messages in the group are available.

MQC.MQGMO_ALL_SEGMENTS_AVAILABLE*

Retrieve the segments of a logical message only when all the segments in the group are available.

MQC.MQGMO_BROWSE_FIRST

Browse from start of queue.

MQC.MQGMO_BROWSE_MSG_UNDER_CURSOR*

Browse message under browse cursor.

MQC.MQGMO_BROWSE_NEXT

Browse from the current position in the queue.

MQC.MQGMO_COMPLETE_MSG*

Retrieve only complete logical messages.

MQC.MQGMO_CONVERT

Request the application data to be converted, to conform to the CharSet and Encoding attributes of the MQMessage, before the data is copied into the message buffer. Because data conversion is also applied when the data is retrieved from the message buffer, applications do not set this option.

Using this option can cause problems when converting from single-byte character sets to double-byte character sets. Instead, do the conversion using the `readString`, `readLine`, and `writeString` methods after the message has been delivered.

MQC.MQGMO_FAIL_IF_QUIESCING

Fail if the queue manager is quiescing.

MQC.MQGMO_LOCK*

Lock the message that is browsed.

MQC.MQGMO_LOGICAL_ORDER*

Return messages in groups, and segments of logical messages, in logical order.

If you use the `MQGMO_LOGICAL_ORDER` option in a reconnectable client, the `MQRC_RECONNECT_INCOMPATIBLE` reason code is returned to the application.

MQC.MQGMO_MARK_SKIP_BACKOUT*

Allow a unit of work to be backed out without reinstating the message on the queue.

MQC.MQGMO_MSG_UNDER_CURSOR

Get message under browse cursor.

MQC.MQGMO_NONE

No other options have been specified; all options assume their default values.

MQC.MQGMO_NO_PROPERTIES

No properties of the message, except properties contained in the message descriptor (or extension) are retrieved.

MQC.MQGMO_NO_SYNCPOINT

Get message without sync point control.

MQC.MQGMO_NO_WAIT

Return immediately if there is no suitable message.

MQC.MQGMO_PROPERTIES_AS_Q_DEF

Retrieve message properties as defined by the `PropertyControl` attribute of `MQQueue`. Access to the message properties in the message descriptor, or extension, are not affected by the `PropertyControl` attribute.

MQC.MQGMO_PROPERTIES_COMPATIBILITY

Retrieve message properties with a prefix of `mcd`, `jms`, `usr`, or `mqext`, in `MQRFH2` headers. Other properties of the message, except properties contained in the message descriptor, or extension, are discarded.

MQC.MQGMO_PROPERTIES_FORCE_MQRFH2

Retrieve message properties, except properties contained in the message descriptor, or extension, in `MQRFH2` headers. Use `MQC.MQGMO_PROPERTIES_FORCE_MQRFH2` in applications that are expecting to retrieve properties but cannot be changed to use message handles.

MQC.MQGMO_PROPERTIES_IN_HANDLE

Retrieve message properties using a `MsgHandle`.

MQC.MQGMO_SYNCPOINT

Get the message under sync point control. The message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

MQC.MQGMO_SYNCPOINT_IF_PERSISTENT*

Get message with sync point control if message is persistent.

MQC.MQGMO_UNLOCK*

Unlock a previously locked message.

MQC.MQGMO_WAIT

Wait for a message to arrive.

public string ResolvedQueueName {get;}

The queue manager sets ResolvedQueueName to the local name of the queue from which the message was retrieved. ResolvedQueueName is different from the name used to open the queue if an alias queue or model queue was opened.

public char Segmentation {get;}*

Segmentation indicates whether you can allow segmentation for the retrieved message. Possible values are:

MQC.MQSEG_INHIBITED

Do not allow segmentation.

MQC.MQSEG_ALLOWED

Allow segmentation

public byte SegmentStatus {get;}*

SegmentStatus is an output field that indicates whether the retrieved message is a segment of a logical message. If the message is a segment, the flag indicates whether it is the last segment. Possible values are:

MQC.MQSS_LAST_SEGMENT

Message is the last or only segment of the logical message.

MQC.MQSS_NOT_A_SEGMENT

Message is not a segment.

MQC.MQSS_SEGMENT

Message is a segment, but is not the last segment of the logical message.

public int WaitInterval {get; set;}

WaitInterval is the maximum time in milliseconds that an MQQueue.get call waits for a suitable message to arrive. Use WaitInterval with MQC.MQGMO_WAIT. Set a value of MQC.MQWI_UNLIMITED to wait an unlimited time for a message.

Constructors

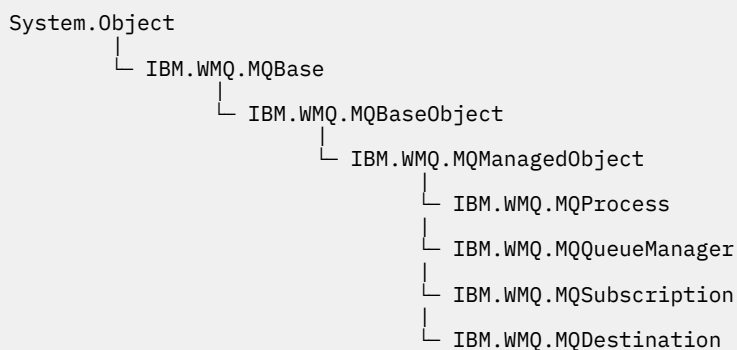
public MQGetMessageOptions()

Construct a new MQGetMessageOptions object with Options set to MQC.MQGMO_NO_WAIT, WaitInterval set to zero, and ResolvedQueueName set to blank.

MQManagedObject.NET class

Use MQManagedObject to inquire and set attributes of MQDestination, MQProcess, MQQueueManager, and MQSubscription. MQManagedObject is a superclass of these classes.

Classes



public class IBM.WMQ.MQManagedObject extends IBM.WMQ.MQBaseObject;

- [“Properties” on page 1771](#)

- [“Methods” on page 1771](#)
- [“Constructors” on page 1773](#)

Properties

Test for `MQException` being thrown when getting properties.

public string AlternateUserId {get; set;}

The alternate user ID, if any, set when the resource was opened. `AlternateUserId.set` is ignored when issued for an object that is opened. `AlternateUserId` is not valid for subscriptions.

public int CloseOptions {get; set;}

Set this attribute to control the way the resource is closed. The default value is `MQC.MQCO_NONE`. `MQC.MQCO_NONE` is the only permissible value for all resources other than permanent dynamic queues, temporary dynamic queues, subscriptions, and topics that are being accessed by the objects that created them.

For queues and topics, the following additional values are permissible:

MQC.MQCO_DELETE

Delete the queue if there are no messages.

MQC.MQCO_DELETE_PURGE

Delete the queue, purging any messages on it.

MQC.MQCO_QUIESCE

Request the queue be closed, receiving a warning if any messages remain (allowing them to be retrieved before final closing).

For subscriptions, the following additional values are permissible:

MQC.MQCO_KEEP_SUB

The subscription is not deleted. This option is valid only if the original subscription is durable. `MQC.MQCO_KEEP_SUB` is the default value for a durable topic.

MQC.MQCO_REMOVE_SUB

The subscription is deleted. `MQC.MQCO_REMOVE_SUB` is the default value for a non-durable, unmanaged topic.

MQC.MQCO_PURGE_SUB

The subscription is deleted. `MQC.MQCO_PURGE_SUB` is the default value for a non-durable, managed topic.

public MQQueueManager ConnectionReference {get;}

The queue manager to which this resource belongs.

public string MQDescription {get;}

The description of the resource as held by the queue manager. `MQDescription` returns an empty string for subscriptions and topics.

public boolean IsOpen {get;}

Indicates whether the resource is currently open.

public string Name {get;}

The name of the resource. The name is either the supplied on the access method, or the allocated by the queue manager for a dynamic queue.

public int OpenOptions {get; set;}

`OpenOptions` are set when an IBM MQ object is opened. The `OpenOptions.set` method is ignored and does not cause an error. Subscriptions have no `OpenOptions`.

Methods

public virtual void Close();

Throws `MQException`.

Closes the object. No further operations against this resource are permitted after calling Close. To change the behavior of the Close method, set the `closeOptions` attribute.

public string GetAttributeString(int *selector*, int *length*);

Throws MQException.

Gets an attribute string.

selector

Integer indicating which attribute is being queried.

length

Integer indicating the length of the string required.

public void Inquire(int[] *selectors*, int[] *intAttrs*, byte[] *charAttrs*);

Throws MQException.

Returns an array of integers and a set of character strings containing the attributes of a queue, process, or queue manager. The attributes to be queried are specified in the *selectors* array.

Note: Many of the more common attributes can be queried using the Get methods defined in MQManagedObject, MQQueue and MQQueueManager.

selectors

Integer array identifying the attributes with values to be inquired on.

intAttrs

The array in which the integer attribute values are returned. Integer attribute values are returned in the same order as the integer attribute selectors in the *selectors* array.

charAttrs

The buffer in which the character attributes are returned, concatenated. Character attributes are returned in the same order as the character attribute selectors in the *selectors* array. The length of each attribute string is fixed for each attribute.

public void Set(int[] *selectors*, int[] *intAttrs*, byte[] *charAttrs*);

Throws MQException.

Sets the attributes defined in the vector of selectors. The attributes to be set are specified in the *selectors* array.

selectors

Integer array identifying the attributes with values to be set.

intAttrs

The array of integer attribute values to be set. These values must be in the same order as the integer attribute selectors in the *selectors* array.

charAttrs

The buffer in which the character attributes to be set are concatenated. These values must be in the same order as the character attribute selectors in the *selectors* array. The length of each character attribute is fixed.

public void SetAttributeString(int *selector*, string *value*, int *length*);

Throws MQException.

Sets an attribute string.

selector

Integer indicating which attribute is being set.

value

The string to set as the attribute value.

length

Integer indicating the length of the string required.

Constructors

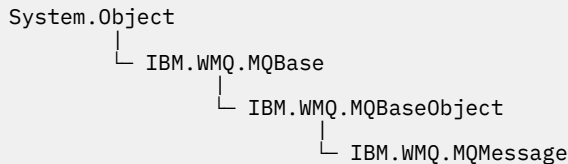
protected MQManagedObject()

Constructor method. This object is an abstract base class which cannot be instantiated by itself.

MQMessage.NET class

Use MQMessage to access the message descriptor and data for an IBM MQ message. MQMessage encapsulates an IBM MQ message.

Class



```
public class IBM.WMQ.MQMessage extends IBM.WMQ.MQBaseObject;
```

Create an MQMessage object and then use the Read and Write methods to transfer data between the message and other objects in your application. Send and receive MQMessage objects using the Put and Get methods of the MQDestination, MQQueue and MQTopic classes.

Get and set the properties of the message descriptor using the properties of MQMessage. Set and Get extended message properties using the SetProperty and GetProperty methods.

- [“Properties” on page 1773](#)
- [“Read and Write message methods” on page 1779](#)
- [“Buffer methods” on page 1781](#)
- [“Property methods” on page 1782](#)
- [“Constructors” on page 1784](#)

Properties

Test for MQException being thrown when getting properties.

public string AccountingToken {get; set;}

Part of the identity context of the message; it helps an application to charge for work done as a result of the message. The default value is MQC.MQACT_NONE.

public string ApplicationIdData {get; set;}

Part of the identity context of the message. ApplicationIdData is information that is defined by the application suite, and can be used to provide additional information about the message or its originator. The default value is "".

public string ApplicationOriginData {get; set;}

Information defined by the application that can be used to provide additional information about the origin of the message. The default value is "".

public int BackoutCount {get;}

A count of the number of times the message has previously been returned and backed out by an MQQueue.Get call as part of a unit of work. The default value is zero.

public int CharacterSet {get; set;}

The coded character set identifier of character data in the message.

Set CharacterSet to identify the character set of character data in the message. Get CharacterSet to find out in what character set has been used to encode character data in the message.

.NET applications always run in Unicode, whereas in other environments applications run in the same character set as the queue manager is running under.

The `ReadString` and `ReadLine` methods convert the character data in the message to Unicode for you.

The `WriteString` method converts from Unicode to the character set encoded in `CharacterSet`. If `CharacterSet` is set to its default value, `MQC.MQCCSI_Q_MGR`, which is 0, no conversion takes place and `CharacterSet` is set to 1200. If you set `CharacterSet` to some other value, `WriteString` converts from Unicode to the alternate value.

Note: Other read and write methods do not use `CharacterSet`.

- `ReadChar` and `WriteChar` read and write a Unicode character to and from the message buffer without conversion.
- `ReadUTF` and `WriteUTF` convert between a Unicode string in the application, and a UTF-8 string, prefixed by a 2-byte length field, in the message buffer.
- Byte methods transfer bytes between the application and the message buffer without alteration.

public byte[] CorrelationId {get; set;}

- For an `MQQueue.Get` call, the correlation identifier of the message to be retrieved. The queue manager returns the first message with a message identifier and a correlation identifier that match the message descriptor fields. The default value, `MQC.MQCI_NONE`, helps any correlation identifier to match.
- For an `MQQueue.Put` call, the correlation identifier to set.

public int DataLength {get;}

The number of bytes of message data remaining to be read.

public int DataOffset {get; set;}

The current cursor position within the message data. Reads and writes take effect at the current position.

public int Encoding {get; set;}

The representation used for numeric values in the application message data. Encoding applies to binary, packed decimal, and floating point data. The behavior of the read and write methods for these numeric formats is altered accordingly. Construct a value for the encoding field by adding one value from each of these three sections. Alternatively, construct the value combining the values from each of the three sections using the bitwise OR operator.

1. Binary integer

MQC.MQENC_INTEGER_NORMAL

Big-endian integers.

MQC.MQENC_INTEGER_REVERSED

Little-endian integers, as used in Intel architecture.

2. Packed-decimal

MQC.MQENC_DECIMAL_NORMAL

Big-endian packed-decimal, as used by z/OS.

MQC.MQENC_DECIMAL_REVERSED

Little-endian packed-decimal.

3. Floating-point

MQC.MQENC_FLOAT_IEEE_NORMAL

Big-endian IEEE floats.

MQC.MQENC_FLOAT_IEEE_REVERSED

Little-endian IEEE floats, as used Intel architecture.

MQC.MQENC_FLOAT_S390

z/OS format floating points.

The default value is:

```
MQC.MQENC_INTEGER_REVERSED |  
MQC.MQENC_DECIMAL_REVERSED |  
MQC.MQENC_FLOAT_IEEE_REVERSED
```

The default setting causes `WriteInt` to write a little-endian integer, and `ReadInt` to read a little-endian integer. If you set the flag `MQC.MQENC_INTEGER_NORMAL` flag instead, `WriteInt` writes a big-endian integer, and `ReadInt` reads a big-endian integer.

Note: A loss in precision can occur when converting from IEEE format floating points to zSeries format floating points.

public int Expiry {get; set;}

An expiry time expressed in tenths of a second, set by the application that puts the message. After the expiry time of a message has elapsed, it is eligible to be discarded by the queue manager. If the message specified one of the `MQC.MQRO_EXPIRATION` flags, a report is generated when the message is discarded. The default value is `MQC.MQEI_UNLIMITED`, meaning that the message never expires.

public int Feedback {get; set;}

Use `Feedback` with a message of type `MQC.MQMT_REPORT` to indicate the nature of the report. The following feedback codes are defined by the system:

- `MQC.MQFB_EXPIRATION`
- `MQC.MQFB_COA`
- `MQC.MQFB_COD`
- `MQC.MQFB_QUIT`
- `MQC.MQFB_PAN`
- `MQC.MQFB_NAN`
- `MQC.MQFB_DATA_LENGTH_ZERO`
- `MQC.MQFB_DATA_LENGTH_NEGATIVE`
- `MQC.MQFB_DATA_LENGTH_TOO_BIG`
- `MQC.MQFB_BUFFER_OVERFLOW`
- `MQC.MQFB_LENGTH_OFF_BY_ONE`
- `MQC.MQFB_IIH_ERROR`

Application-defined feedback values in the range `MQC.MQFB_APPL_FIRST` to `MQC.MQFB_APPL_LAST` can also be used. The default value of this field is `MQC.MQFB_NONE`, indicating that no feedback is provided.

public string Format {get; set;}

A format name used by the sender of the message to indicate the nature of the data in the message to the receiver. You can use your own format names, but names beginning with the letters `MQ` have meanings that are defined by the queue manager. The queue manager built-in formats are:

MQC.MQFMT_ADMIN

Command server request/reply message.

MQC.MQFMT_COMMAND_1

Type 1 command reply message.

MQC.MQFMT_COMMAND_2

Type 2 command reply message.

MQC.MQFMT_DEAD_LETTER_HEADER

Dead-letter header.

MQC.MQFMT_EVENT

Event message.

MQC.MQFMT_NONE

No format name.

MQC.MQFMT_PCF

User-defined message in programmable command format.

MQC.MQFMT_STRING

Message consisting entirely of characters.

MQC.MQFMT_TRIGGER

Trigger message

MQC.MQFMT_XMIT_Q_HEADER

Transmission queue header.

The default value is MQC.MQFMT_NONE.

public byte[] GroupId {get; set;}

A byte string that identifies the message group to which the physical message belongs. The default value is MQC.MQGI_NONE.

public int MessageFlags {get; set;}

Flags controlling the segmentation and status of a message.

public byte[] MessageId {get; set;}

For an MQQueue.Get call, this field specifies the message identifier of the message to be retrieved. Normally, the queue manager returns the first message with a message identifier and correlation identifier that match the message descriptor fields. Allow any message identifier to match using the special value MQC.MQMI_NONE.

For an MQQueue.Put call, this field specifies the message identifier to use. If MQC.MQMI_NONE is specified, the queue manager generates a unique message identifier when the message is put. The value of this member variable is updated after the put, to indicate the message identifier that was used. The default value is MQC.MQMI_NONE.

public int MessageLength {get;}

The number of bytes of message data in the MQMessage object.

public int MessageSequenceNumber {get; set;}

The sequence number of a logical message within a group.

public int MessageType {get; set;}

Indicates the type of the message. The following values are currently defined by the system:

- MQC.MQMT_DATAGRAM
- MQC.MQMT_REPLY
- MQC.MQMT_REPORT
- MQC.MQMT_REQUEST

Application-defined values can also be used, in the range MQC.MQMT_APPL_FIRST to MQC.MQMT_APPL_LAST. The default value of this field is MQC.MQMT_DATAGRAM.

public int Offset {get; set;}

In a segmented message, the offset of data in a physical message from the start of a logical message.

public int OriginalLength {get; set;}

The original length of a segmented message.

public int Persistence {get; set;}

Message persistence. The following values are defined:

- MQC.MQPER_NOT_PERSISTENT

If you set this option in a reconnectable client, the MQRC_NONE reason code is returned to the application when the connection is successful.

- MQC.MQPER_PERSISTENT

If you set this option in a reconnectable client, the MQRC_CALL_INTERRUPTED reason code is returned to the application after the connection is successful.

- MQC.MQPER_PERSISTENCE_AS_Q_DEF

The default value is MQC.MQPER_PERSISTENCE_AS_Q_DEF, which takes the persistence for the message from the default persistence attribute of the destination queue.

public int Priority {get; set;}

The message priority. The special value MQC.MQPRI_PRIORITY_AS_Q_DEF can also be set in outbound message. The priority for the message is then taken from the default priority attribute of the destination queue. The default value is MQC.MQPRI_PRIORITY_AS_Q_DEF.

public int PropertyValidation {get; set;}

Specifies whether validation of properties takes place when a property of the message is set. Possible values are:

- MQCMHO_DEFAULT_VALIDATION
- MQCMHO_VALIDATE
- MQCMHO_NO_VALIDATION

The default value is MQCMHO_DEFAULT_VALIDATION.

public string PutApplicationName {get; set;}

The name of the application that put the message. The default value is "".

public int PutApplicationType {get; set;}

The type of application that put the message. PutApplicationType can be a system-defined or user-defined value. The following values are defined by the system:

- MQC.MQAT_AIX
- MQC.MQAT_CICS
- MQC.MQAT_DOS
- MQC.MQAT_IMS
- MQC.MQAT_MVS
- MQC.MQAT_OS2
- MQC.MQAT_OS400
- MQC.MQAT_QMGR
- MQC.MQAT_UNIX
- MQC.MQAT_WINDOWS
- MQC.MQAT_JAVA

The default value is MQC.MQAT_NO_CONTEXT, which indicates that no context information is present in the message.

public DateTime PutDateTime {get; set;}

The time and date that the message was put.

public string ReplyToQueueManagerName {get; set;}

The name of the queue manager to send reply or report messages. The default value is "", and the queue manager provides the ReplyToQueueManagerName.

public string ReplyToQueueName {get; set;}

The name of the message queue to which the application that issued the get request for the message sends MQC.MQMT_REPLY and MQC.MQMT_REPORT messages. The default ReplyToQueueName is "".

public int Report {get; set;}

Use Report to specify options about report and reply messages:

- Whether reports are required.
- Whether the application message data is to be included in the reports.
- How to set the message and correlation identifiers in the report or reply.

Any combination of the four report types can be requested:

- Specify any combination of the four report types. Selecting any of the three options for each report type, depending on whether the application message data is to be included in the report message.

1. Confirm on arrival

- MQC.MQRO_COA
- MQC.MQRO_COA_WITH_DATA
- MQC.MQRO_COA_WITH_FULL_DATA **

2. Confirm on delivery

- MQC.MQRO_COD
- MQC.MQRO_COD_WITH_DATA
- MQC.MQRO_COD_WITH_FULL_DATA **

3. Exception

- MQC.MQRO_EXCEPTION
- MQC.MQRO_EXCEPTION_WITH_DATA
- MQC.MQRO_EXCEPTION_WITH_FULL_DATA **

4. Expiration

- MQC.MQRO_EXPIRATION
- MQC.MQRO_EXPIRATION_WITH_DATA
- MQC.MQRO_EXPIRATION_WITH_FULL_DATA **

Note: Values marked with ** in the list are not supported by z/OS queue managers. Do not use them if your application is likely to access a z/OS queue manager, regardless of the platform on which the application is running.

- Specify one of the following to control how the message ID is generated for the report or reply message:

- MQC.MQRO_NEW_MSG_ID
- MQC.MQRO_PASS_MSG_ID

- Specify one of the following to control how the correlation ID of the report or reply message is to be set:

- MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQC.MQRO_PASS_CORREL_ID

- Specify one of the following to control the disposition of the original message when it cannot be delivered to the destination queue:

- MQC.MQRO_DEAD_LETTER_Q
- MQC.MQRO_DISCARD_MSG **

- If no report options are specified, the default is:

```
MQC.MQRO_NEW_MSG_ID |
MQC.MQRO_COPY_MSG_ID_TO_CORREL_ID |
MQC.MQRO_DEAD_LETTER_Q
```

- You can specify one or both of the following to request that the receiving application sends a positive action or negative action report message.

- MQC.MQRO_PAN
- MQC.MQRO_NAN

public int TotalMessageLength {get;}

The total number of bytes in the message as stored on the message queue from which this message was received.

public string UserId {get; set;}

UserId is part of the identity context of the message. The queue manager generally provides the value. You can override the value if you have authority to set the identity context.

public int Version {get; set;}

The version of the MQMD structure in use.

Read and Write message methods

The Read and Write methods perform the same functions as the members of the `BinaryReader` and `BinaryWriter` classes in the `.NET System.IO` namespace. See MSDN for the full language syntax and usage examples. The methods read or write from the current position in the message buffer. They move the current position forward by the number of bytes read or written.

Note: If the message data contains an MQRFH or MQRFH2 header, you must use the `ReadBytes` method to read the data.

- All the methods throw `IOException`.
- The `ReadFully` methods automatically resize the target byte or sbyte array to fit the message exactly. A null array is also resized.
- Read methods throw `EndOfStreamException`.
- `WriteDecimal` methods throw `MQException`.
- `ReadString`, `ReadLine` and `WriteString` methods convert between Unicode and the character set of the message; see [CharacterSet](#).
- The `Decimal` methods read and write packed decimal numbers encoded either in big-endian, `MQC.MQENC_DECIMAL_NORMAL`, or little-endian `MQC.MQENC_DECIMAL_REVERSE` format, according to the value of `Encoding`. Decimal ranges and corresponding .NET types are as follows:

Decimal2/short

-999 to 999

Decimal4/int

-9999999 to 9999999

Decimal8/long

-999999999999999 to 999999999999999

- The `Double` and `Float` methods read and write floating values encoded in IEEE big-endian and little-endian formats, `MQC.MQENC_FLOAT_IEEE_NORMAL` and `MQC.MQENC_FLOAT_IEEE_REVERSED`, or in S/390 format, `MQC.MQENC_FLOAT_S390`, according to the value of `Encoding`.
- The `Int` methods read and write integer values encoded in big-endian, `MQC.MQENC_INTEGER_NORMAL`, or little-endian, `MQC.MQENC_INTEGER_REVERSED`, format, according to the value of `Encoding`. The integers are all signed, except for the addition of an unsigned 2-byte integer type. The integer sizes, and .NET and IBM MQ types are as follows:

2 byte

short, Int2, ushort, UInt2

4 byte

int, Int4

8 byte

long, Int8

- `WriteObject` transfers the class of an object, the values of its non-transient and non-static fields, and the fields of its supertypes, to the message buffer.
- `ReadObject` creates an object from the class of the object, the signature of the class, and the values of its non-transient and non-static fields, and the fields of its supertypes.

Table 670. Read and Write message methods

Target type	Method signatures
Boolean	<pre>public bool ReadBoolean();</pre> <pre>public void WriteBoolean(bool value);</pre>
Byte	<pre>public byte ReadByte()</pre> <pre>public byte ReadUnsignedByte()</pre> <pre>public void Write(int value)</pre> <pre>public void WriteByte(int value)</pre> <pre>public void WriteByte(byte value)</pre> <pre>public void WriteByte(sbyte value)</pre>
Bytes	<pre>public byte[] ReadBytes(int count)</pre> <pre>public void ReadFully(ref byte[] value)</pre> <pre>public void ReadFully(ref sbyte[] value)</pre> <pre>public void ReadFully(ref byte[] value, int offset, int length)</pre> <pre>public void ReadFully(ref sbyte[] value, int offset, int length)</pre> <pre>public void Write(byte[] value)</pre> <pre>public void Write(sbyte[] value)</pre> <pre>public void Write(byte[] value, int offset, int length)</pre> <pre>public void Write(sbyte[] value, int offset, int length)</pre> <pre>public void WriteBytes(string value)</pre>
Decimal2	<pre>public void WriteDecimal2(short value)</pre>
Decimal4	<pre>public void WriteDecimal4(short value)</pre>
Decimal8	<pre>public void WriteDecimal8(short value)</pre>
Double	<pre>public double ReadDouble()</pre> <pre>public void WriteDouble(double value)</pre>
Float	<pre>public float ReadFloat()</pre> <pre>public void WriteFloat(float value)</pre>
Int2	<pre>public void WriteInt2(int value)</pre>
Int4	<pre>public int readDecimal4()</pre> <pre>public int ReadInt()</pre> <pre>public int ReadInt4()</pre> <pre>public void WriteInt(int value)</pre> <pre>public void WriteInt4(int value)</pre>
Int8	<pre>public void WriteInt8(long value)</pre>

Table 670. Read and Write message methods (continued)

Target type	Method signatures
Long	<pre>public long ReadDecimal8() public long ReadLong() public long ReadInt8() public void WriteLong(long value)</pre>
Object	<pre>public Object ReadObject() public void WriteObject(Object object)</pre>
Short	<pre>public short ReadShort() public short ReadDecimal2() public short ReadInt2() public void WriteShort(int value)</pre>
string	<pre>public string ReadString(int length) public void WriteString(string string)</pre>
Unsigned Short	<pre>public ushort ReadUnsignedShort() public ushort ReadUInt2()</pre>
Unicode	<pre>public string ReadLine() public char ReadChar() public void WriteChar(int value) public void WriteChars(string string)</pre>
UTF	<pre>public string ReadUTF() public void WriteUTF(string string)</pre>

Buffer methods

public void ClearMessage();

Throws IOException.

Discards any data in the message buffer and sets the data offset back to zero.

public void ResizeBuffer(int size)

Throws IOException.

A hint to the MQMessage object about the size of buffer that might be required for subsequent get operations. If the message currently contains message data, and the new size is less than the current size, the message data is truncated.

public void Seek(int pos)

Throws IOException, ArgumentOutOfRangeException, ArgumentException.

Moves the cursor to the absolute position in the message buffer given by *pos*. Subsequent reads and writes act at this position in the buffer.

public int SkipBytes(int i)

Throws IOException, EndOfStreamException.

Moves forward *n* bytes in the message buffer and returns *n*, the number of bytes skipped.

SkipBytes method blocks until one of the following events occurs:

- All the bytes are skipped
- The end of message buffer is detected
- An exception is thrown

Property methods

public void DeleteProperty(string name);

Throws MQException.

Deletes a property with the specified name from the message.

name

The name of the property to delete.

public System.Collections.IEnumerator GetPropertyNames(string name)

Throws MQException.

Returns an IEnumerator of all the property names matching the specified name. The percent sign '%' can be used at the end of the name as a wildcard character to filter the properties of the message, matching on zero, or more characters, including the period.

name

The name of the property to match on.

SetProperty and GetProperty methods

All the SetProperty and GetProperty methods throw MQException.

The SetProperty method of MQMessage .NET class adds a new property if a property does not exist already. However, if the property already exists, then the supplied property value is added to the end of the list. When multiple values are set to a property name using SetProperty, calling GetProperty for that name returns those values sequentially in the order that those values were set.

The behavior is same for all the Set*Property and Get*Property typed methods such as GetLongProperty, SetLongProperty, GetBooleanProperty, SetBooleanProperty, GetStringProperty, and SetStringProperty.

Table 671. SetProperty and GetProperty methods	
Type	Method signatures
Boolean	<pre>public boolean GetBooleanProperty(string name); public boolean GetBooleanProperty(string name, MQPropertyDescriptor pd);</pre>
	<pre>public void SetBooleanProperty(string name, boolean value); public void SetBooleanProperty(string name, MQPropertyDescriptor pd, boolean value);</pre>
Byte	<pre>public sbyte GetByteProperty(string name); public sbyte GetByteProperty(string name, MQPropertyDescriptor pd);</pre>
	<pre>public void SetByteProperty(string name, sbyte value); public void SetByteProperty(string name, MQPropertyDescriptor pd, sbyte value);</pre>

Table 671. *SetProperty* and *GetProperty* methods (continued)

Type	Method signatures
Bytes	<pre>public sbyte[] GetBytesProperty(string name); public sbyte[] GetBytesProperty(string name, MQPropertyDescriptor pd); public void SetBytesProperty(string name, sbyte[] value); public void SetBytesProperty(string name, MQPropertyDescriptor pd, sbyte[] value);</pre>
Double	<pre>public double GetDoubleProperty(string name); public double GetDoubleProperty(string name, MQPropertyDescriptor pd); public void SetDoubleProperty(string name, double value); public void SetDoubleProperty(string name, MQPropertyDescriptor pd, double value);</pre>
Float	<pre>public float GetFloatProperty(string name); public float GetFloatProperty(string name, MQPropertyDescriptor pd); public void SetFloatProperty(string name, float value); public void SetFloatProperty(string name, MQPropertyDescriptor pd, float value);</pre>
Int2	<pre>public short GetInt2Property(string name); public short GetInt2Property(string name, MQPropertyDescriptor pd); public void SetInt2Property(string name, short value); public void SetInt2Property(string name, MQPropertyDescriptor pd, short value);</pre>
Int4	<pre>public int GetInt4Property(string name); public int GetInt4Property(string name, MQPropertyDescriptor pd); public void SetInt4Property(string name, int value); public void SetInt4Property(string name, MQPropertyDescriptor pd, int value);</pre>
Int8	<pre>public long GetInt8Property(string name); public long GetInt8Property(string name, MQPropertyDescriptor pd); public void SetInt8Property(string name, long value); public void SetInt8Property(string name, MQPropertyDescriptor pd, long value);</pre>
Long	<pre>public long GetLongProperty(string name); public long GetLongProperty(string name, MQPropertyDescriptor pd); public void SetLongProperty(string name, long value); public void SetLongProperty(string name, MQPropertyDescriptor pd, long value);</pre>
Object	<pre>public Object GetObjectProperty(string name); public Object GetObjectProperty(string name, MQPropertyDescriptor pd); public void SetObjectProperty(string name, Object value); public void SetObjectProperty(string name, MQPropertyDescriptor pd, Object value);</pre>

Table 671. *SetProperty* and *GetProperty* methods (continued)

Type	Method signatures
Short	<pre>public short GetShortProperty(string name); public short GetShortProperty(string name, MQPropertyDescriptor pd); public void SetShortProperty(string name, short value); public void SetShortProperty(string name, MQPropertyDescriptor pd, short value);</pre>
string	<pre>public string GetStringProperty(string name); public string GetStringProperty(string name, MQPropertyDescriptor pd); public void SetStringProperty(string name, string value); public void SetStringProperty(string name, MQPropertyDescriptor pd, string value);</pre>

Constructors

public MQMessage();

Creates an MQMessage object with default message descriptor information and an empty message buffer.

MQProcess.NET class

Use MQProcess to query the attributes of an IBM MQ process. Create an MQProcess object using a constructor, or an MQQueueManager AccessProcess method.

Class

```
System.Object
├── IBM.WMQ.MQBase
│   └── IBM.WMQ.MQBaseObject
│       └── IBM.WMQ.MQManagedObject
│           └── IBM.WMQ.MQProcess
```

```
public class IBM.WMQ.MQProcess extends IBM.WMQ.MQManagedObject;
```

- [“Properties” on page 1784](#)
- [“Constructors” on page 1785](#)

Properties

Test for MQException being thrown when getting properties.

public string ApplicationId {get;}

Gets the character string that identifies the application to be started. ApplicationId is used by a trigger monitor application. ApplicationId is sent to the initiation queue as part of the trigger message.

The default value is null.

public int ApplicationType {get;}

Identifies the type of the process to be started by a trigger monitor application. Standard types are defined, but others can be used:

- MQAT_AIX

- MQAT_CICS
- MQAT_IMS
- MQAT_MVS
- MQAT_NATIVE
- MQAT_OS400
- MQAT_UNIX
- MQAT_WINDOWS
- MQAT_JAVA
- MQAT_USER_FIRST
- MQAT_USER_LAST

The default value is MQAT_NATIVE.

public string EnvironmentData {get;}

Gets information about the environment of the application that is to be started.

The default value is null.

public string UserData {get;}

Gets information the user has provided about the application to be started.

The default value is null.

Constructors

public MQProcess(MQQueueManager queueManager, string processName, int openOptions);
public MQProcess(MQQueueManager qMgr, string processName, int openOptions, string queueManagerName, string alternateUserId);

Throws MQException.

Access an IBM MQ process on queue manager *qMgr* to inquire on process attributes.

qMgr

Queue manager to access.

processName

The name of the process to open.

openOptions

Options that control the opening of the process. The valid options that can be added, or combined using a bitwise OR, are:

- MQC.MQ00_FAIL_IF QUIESCING
- MQC.MQ00_INQUIRE
- MQC.MQ00_SET
- MQC.MQ00_ALTERNATE_USER_AUTHORITY

queueManagerName

The name of the queue manager on which the process is defined. You can leave a blank or null queue manager name if the queue manager is the same as the one the process is accessing.

alternateUserId

If MQC.MQ00_ALTERNATE_USER_AUTHORITY is specified in the *openOptions* parameter, *alternateUserId* specifies the alternative user ID used to check the authorization for the action. If MQ00_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be blank or null.

Default user authority is used for connection to the queue manager if MQC.MQ00_ALTERNATE_USER_AUTHORITY is not specified.

```
public MQProcess MQQueueManager.AccessProcess(string processName, int  
openOptions);  
public MQProcess MQQueueManager.AccessProcess(string processName, int  
openOptions, string queueManagerName, string alternateUserId);
```

Throws MQException.

Access an IBM MQ process on this queue manager to inquire on process attributes.

processName

The name of the process to open.

openOptions

Options that control the opening of the process. The valid options that can be added, or combined using a bitwise OR, are:

- MQC.MQ00_FAIL_IF QUIESCING
- MQC.MQ00_INQUIRE
- MQC.MQ00_SET
- MQC.MQ00_ALTERNATE_USER_AUTHORITY

queueManagerName

The name of the queue manager on which the process is defined. You can leave a blank or null queue manager name if the queue manager is the same as the one the process is accessing.

alternateUserId

If MQC.MQ00_ALTERNATE_USER_AUTHORITY is specified in the *openOptions* parameter, *alternateUserId* specifies the alternative user ID used to check the authorization for the action. If MQ00_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be blank or null.

Default user authority is used for connection to the queue manager if MQC.MQ00_ALTERNATE_USER_AUTHORITY is not specified.

MQPropertyDescriptor.NET class

Use MQPropertyDescriptor as a parameter to MQMessage GetProperty and SetProperty methods. MQPropertyDescriptor describes an MQMessage property.

Class

```
System.Object  
└─ IBM.WMQ.MQPropertyDescriptor
```

```
public class IBM.WMQ.MQPropertyDescriptor extends System.Object;
```

- [“Properties” on page 1786](#)
- [“Constructors” on page 1788](#)

Properties

Test for MQException being thrown when getting properties.

```
public int Context {get; set;}
```

The message context the property belongs to. Possible values are:

MQC.MQPD_NO_CONTEXT

The property is not associated with a message context.

MQC.MQPD_USER_CONTEXT

The property is associated with the user context.

If the user is authorized, a property associated with the user context is saved when a message is retrieved. A subsequent Put method referencing the saved context, can pass the property into the new message.

public int CopyOptions {get; set;}

CopyOptions describes which type of message the property can be copied into.

When a queue manager receives a message containing an IBM MQ defined property that the queue manager recognizes as being incorrect, the queue manager corrects the value of the CopyOptions field.

Any combination of the following options can be specified. Combine the options by adding the values, or using bitwise OR.

MQC.MQCOPY_ALL

The property is copied into all types of subsequent messages.

MQC.MQCOPY_FORWARD

The property is copied into a message being forwarded.

MQC.MQCOPY_PUBLISH

The property is copied into the message received by a subscriber when a message is being published.

MQC.MQCOPY_REPLY

The property is copied into a reply message.

MQC.MQCOPY_REPORT

The property is copied into a report message.

MQC.MQCOPY_DEFAULT

The value indicated no other copy options have been specified. No relationship exists between the property and subsequent messages. MQC.MQCOPY_DEFAULT is always returned for message descriptor properties.

MQC.MQCOPY_NONE

The same as MQC.MQCOPY_DEFAULT

public int Options { set; }

Options defaults to CMQC.MQPD_NONE. You cannot set any other value.

public int Support { get; set; }

Set Support to specify the level of support required for IBM MQ-defined message properties. Support for all other properties is optional. Any or none of the following values can be specified

MQC.MQPD_SUPPORT_OPTIONAL

The property is accepted by a queue manager even if it is not supported. The property can be discarded in order for the message to flow to a queue manager that does not support message properties. This value is also assigned to properties that are not IBM MQ defined.

MQC.MQPD_SUPPORT_REQUIRED

Support for the property is required. If you put the message to a queue manager that does not support the IBM MQ-defined property, the method fails. It returns completion code MQC.MQCC_FAILED and reason code MQC.MQRC_UNSUPPORTED_PROPERTY.

MQC.MQPD_SUPPORT_REQUIRED_IF_LOCAL

Support for the property is required, if the message is destined for a local queue. If you put the message to a local queue on a queue manager that does not support the IBM MQ-defined property, the method fails. It returns completion code MQC.MQCC_FAILED and reason code MQC.MQRC_UNSUPPORTED_PROPERTY.

No check is made if the message is put to a remote queue manager.

Constructors

PropertyDescriptor();

Create a property descriptor.

MQPutMessageOptions.NET class

Use MQPutMessageOptions to specify how messages are sent. It modifies the behavior of MQDestination.Put.

Class

```
System.Object
├── IBM.WMQ.MQBase
│   └── IBM.WMQ.MQBaseObject
│       └── IBM.WMQ.MQPutMessageOptions
```

```
public class IBM.WMQ.MQPutMessageOptions extends IBM.WMQ.MQBaseObject;
```

- [“Properties” on page 1788](#) [“Constructors” on page 1790](#)

Properties

Test for MQException being thrown when getting properties.

Note: The behavior of some of the options available in this class depends on the environment in which they are used. These elements are marked with an asterisk, *.

public MQQueue ContextReference {get; set;}

If the options field includes MQC.MQPMO_PASS_IDENTITY_CONTEXT or MQC.MQPMO_PASS_ALL_CONTEXT, set this field to refer to the MQQueue from which to take the context information.

The initial value of this field is null.

public int InvalidDestCount {get;} *

Generally, used for distribution lists, InvalidDestCount indicates the number of messages that could not be sent to queues in a distribution list. The count includes queues that failed to open and also the queues that were opened successfully, but for which the put operation had failed.

.NET does not support distribution lists, but InvalidDestCount is set when opening a single queue.

public int KnownDestCount {get;} *

Generally used for distribution lists, KnownDestCount indicates the number of messages that the current call has sent successfully to queues that resolve to local queues.

.NET does not support distribution lists, but InvalidDestCount is set when opening a single queue.

public int Options {get; set;}

Options that control the action of MQDestination.put and MQQueueManager.put. Any or none of the following values can be specified. If more than one option is required, the values can be added or combined using the bitwise OR operator.

MQC.MQPMO_ASYNC_RESPONSE

This option causes the MQDestination.put call to be made asynchronously, with some response data.

MQC.MQPMO_DEFAULT_CONTEXT

Associate default context with the message.

MQC.MQPMO_FAIL_IF QUIESCING

Fail if the queue manager is quiescing.

MQC.MQPMO_LOGICAL_ORDER *

Put logical messages and segments in message groups into their logical order.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned to the application.

MQC.MQPMO_NEW_CORREL_ID *

Generate a new correlation ID for each sent message.

MQC.MQPMO_NEW_MSG_ID *

Generate a new message ID for each sent message.

MQC.MQPMO_NONE

No options specified. Do not use with other options.

MQC.MQPMO_NO_CONTEXT

No context is to be associated with the message.

MQC.MQPMO_NO_SYNCPOINT

Put a message without sync point control. If the sync point control option is not specified, a default of no sync point is assumed.

MQC.MQPMO_PASS_ALL_CONTEXT

Pass all context from an input queue handle.

MQC.MQPMO_PASS_IDENTITY_CONTEXT

Pass identity context from an input queue handle.

MQC.MQPMO_RESPONSE_AS_Q_DEF

For an MQDestination.put call, this option takes the put response type from DEFPRESP attribute of the queue.

For an MQQueueManager.put call, this option causes the call to be made synchronously.

MQC.MQPMO_RESPONSE_AS_TOPIC_DEF

MQC.MQPMO_RESPONSE_AS_TOPIC_DEF is a synonym for MQC.MQPMO_RESPONSE_AS_Q_DEF for use with topic objects.

MQC.MQPMO_RETAIN

The publication being sent is to be retained by the queue manager. If this option is used and the publication cannot be retained, the message is not published and the call fails with MQC.MQRC_PUT_NOT_RETAINED.

Request a copy of this publication after the time it was published, by calling the MQSubscription.RequestPublicationUpdate method. The saved publication is sent to applications that create a subscription without setting the MQC.MQSO_NEW_PUBLICATIONS_ONLY option. Check the MQIsRetained message property of a publication, when it is received, to find out if it was the retained publication.

When retained publications are requested by a subscriber, the subscription used might contain a wildcard in the topic string. If there are multiple retained publications in the topic tree that match the subscription, they are all sent.

MQC.MQPMO_SET_ALL_CONTEXT

Set all context from the application.

MQC.MQPMO_SET_IDENTITY_CONTEXT

Set identity context from the application.

MQC.MQPMO_SYNC_RESPONSE

This option causes the MQDestination.put or MQQueueManager.put call to be made synchronously, with full response data.

MQC.MQPMO_SUPPRESS_REPLYTO

Any information filled into the ReplyToQueueName and ReplyToQueueManagerName fields of the publication is not passed on to subscribers. If this option is used in combination with a report option that requires a ReplyToQueueName, the call fails with MQC.MQRC_MISSING_REPLY_TO_Q.

MQC.MQPMO_SYNCPOINT

Put a message with sync point control. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted.

public int RecordFields {get; set;} *

Information about distribution lists. Distribution lists are not supporting in .NET.

public string ResolvedQueueManagerName {get;}

An output field set by the queue manager to the name of the queue manager that owns the queue specified by the remote queue name. ResolvedQueueManagerName might be different from the name of the queue manager from which the queue was accessed if the queue is a remote queue.

A nonblank value is returned only if the object is a single queue. If the object is a distribution list or a topic, the value returned is undefined.

public string ResolvedQueueName {get;}

An output field that is set by the queue manager to the name of the queue on which the message is placed. ResolvedQueueName might be different from the name used to open the queue if the opened queue was an alias or model queue.

A non-blank value is returned only if the object is a single queue. If the object is a distribution list or a topic, the value returned is undefined.

public int UnknownDestCount {get;} *

Generally used for distribution lists, UnknownDestCount is an output field set by the queue manager. It reports the number of messages that the current call has sent successfully to queues that resolve to remote queues.

.NET does not support distribution lists, but InvalidDestCount is set when opening a single queue.

Constructors

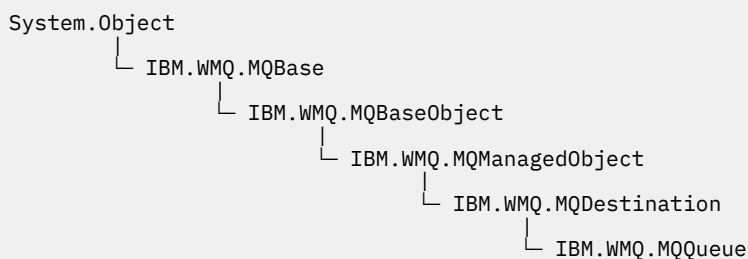
public MQPutMessageOptions();

Construct a new MQPutMessageOptions object with no options set, and a blank ResolvedQueueName and ResolvedQueueManagerName.

MQueue.NET class

Use MQueue to send and receive messages, and query attributes of an IBM MQ queue. Create an MQueue object using a constructor, or an MQueueManager.AccessProcess method.

Class



```
public class IBM.WMQ.MQQueue extends IBM.WMQ.MQDestination;
```

- [“Properties” on page 1791](#)
- [“Methods” on page 1793](#)
- [“Constructors” on page 1795](#)

Properties

Test for MQException being thrown when getting properties.

public int ClusterWorkLoadPriority {get;}

Specifies the priority of the queue. This parameter is valid only for local, remote, and alias queues.

public int ClusterWorkLoadRank {get;}

Specifies the rank of the queue. This parameter is valid only for local, remote, and alias queues.

public int ClusterWorkLoadUseQ {get;}

Specifies the behavior of an MQPUT operation when the target queue has a local instance and at least one remote cluster instance. This parameter does not apply if the MQPUT originates from a cluster channel. This parameter is valid only for local queues.

public DateTime CreationDateTime {get;}

The date and time that this queue was created.

public int CurrentDepth {get;}

Gets the number of messages currently on the queue. This value is incremented during a put call, and during backout of a get call. It is decremented during a non-browse get and during backout of a put call.

public int DefinitionType {get;}

How the queue was defined. The possible values are:

- MQC.MQQDT_PREDEFINED
- MQC.MQQDT_PERMANENT_DYNAMIC
- MQC.MQQDT_TEMPORARY_DYNAMIC

public int InhibitGet {get; set;}

Controls whether you can get messages on this queue or for this topic. The possible values are:

- MQC.MQQA_GET_INHIBITED
- MQC.MQQA_GET_ALLOWED

public int InhibitPut {get; set;}

Controls whether you can put messages on this queue or for this topic. The possible values are:

- MQQA_PUT_INHIBITED
- MQQA_PUT_ALLOWED

public int MaximumDepth {get;}

The maximum number of messages that can exist on the queue at any one time. An attempt to put a message to a queue that already contains this many messages fails with reason code MQC.MQRC_Q_FULL.

public int MaximumMessageLength {get;}

The maximum length of the application data that can exist in each message on this queue. An attempt to put a message larger than this value fails with reason code MQC.MQRC_MSG_TOO_BIG_FOR_Q.

public int NonPersistentMessageClass {get;}

The level of reliability for non-persistent messages put to this queue.

public int OpenInputCount {get;}

The number of handles that are currently valid for removing messages from the queue. OpenInputCount is the total number of valid input handles known to the local queue manager, not just handles created by the application.

public int OpenOutputCount {get;}

The number of handles that are currently valid for adding messages to the queue. OpenOutputCount is the total number of valid output handles known to the local queue manager, not just handles created by the application.

public int QueueAccounting {get;}

Specifies whether you can enable the collection of accounting information for the queue.

public int QueueMonitoring {get;}

Specifies whether you can enable the monitoring for the queue.

public int QueueStatistics {get;}

Specifies whether you can enable the collection of statistics for the queue.

public int QueueType {get;}

The type of this queue with one of the following values:

- MQC.MQQT_ALIAS
- MQC.MQQT_LOCAL
- MQC.MQQT_REMOTE
- MQC.MQQT_CLUSTER

public int Shareability {get;}

Whether the queue can be opened for input multiple times. The possible values are:

- MQC.MQQA_SHAREABLE
- MQC.MQQA_NOT_SHAREABLE

public string TPIPE {get;}

The TPIPE name used for communication with OTMA using the IBM MQ IMS bridge.

public int TriggerControl {get; set;}

Whether trigger messages are written to an initiation queue, to start an application to service the queue. The possible values are:

- MQC.MQTC_OFF
- MQC.MQTC_ON

public string TriggerData {get; set;}

The free-format data that the queue manager inserts into the trigger message. It inserts TriggerData when a message arriving on this queue causes a trigger message to be written to the initiation queue. The maximum permissible length of the string is given by MQC.MQ_TRIGGER_DATA_LENGTH.

public int TriggerDepth {get; set;}

The number of messages that must be on the queue before a trigger message is written when trigger type is set to MQC.MQTT_DEPTH.

public int TriggerMessagePriority {get; set;}

The message priority under which messages do not contribute to the generation of trigger messages. That is, the queue manager ignores these messages when deciding whether to generate a trigger. A value of zero causes all messages to contribute to the generation of trigger messages.

public int TriggerType {get; set;}

The conditions under which trigger messages are written as a result of messages arriving on this queue. The possible values are:

- MQC.MQTT_NONE
- MQC.MQTT_FIRST
- MQC.MQTT EVERY
- MQC.MQTT_DEPTH

Methods

```
public void Get(MQMessage message);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions, int  
MaxMsgSize);
```

Throws MQException.

Gets a message from a queue.

If the get fails, the MQMessage object is unchanged. If it succeeds, the message descriptor and message data portions of the MQMessage are replaced with the message descriptor and message data from the incoming message.

All calls to IBM MQ from a particular MQQueueManager are synchronous. Therefore, if you perform a get with wait, all other threads using the same MQQueueManager are blocked from making further IBM MQ calls until the Get call is accomplished. If you need multiple threads to access IBM MQ simultaneously, each thread must create its own MQQueueManager object.

message

Contains the message descriptor and the returned message data. Some of the fields in the message descriptor are input parameters. It is important to ensure that the MessageId and CorrelationId input parameters are set as required.

A reconnectable client returns the reason code MQRC_BACKED_OUT after successful reconnection, for messages received under MQGM_SYNCPOINT.

getMessageOptions

Options controlling the action of the get.

Using option MQC.MQGM_CONVERT might result in an exception with reason code MQC.MQRC_CONVERTED_STRING_TOO_BIG when converting from single-byte character codes to double byte codes. In this case, the message is copied into the buffer without conversion.

If *getMessageOptions* is not specified, the message option used is MQGM_NOWAIT.

If you use the MQGM_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

MaxMsgSize

The largest message this message object is to receive. If the message on the queue is larger than this size, one of two things occurs:

- If the MQGM_ACCEPT_TRUNCATED_MSG flag is set in the MQGetMessageOptions object, the message is filled with as much of the message data as possible. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_ACCEPTED reason code.
- If the MQGM_ACCEPT_TRUNCATED_MSG flag is not set, the message remains on the queue. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_FAILED reason code.

If *MaxMsgSize* is not specified, the whole message is retrieved.

```
public void Put(MQMessage message);  
public void Put(MQMessage message, MQPutMessageOptions putMessageOptions);
```

Throws MQException.

Puts a message to a queue.

Modifications to the MQMessage object after the Put call has been accomplished do not affect the actual message on the IBM MQ queue or publication topic.

Put updates the `MessageId` and `CorrelationId` properties of the `MQMessage` object and does not clear message data. Further Put or Get calls refer to the updated information in the `MQMessage` object. For example, in the following code snippet, the first message contains a and the second ab.

```
msg.WriteString("a");
q.Put(msg,pmo);
msg.WriteString("b");
q.Put(msg,pmo);
```

message

An `MQMessage` object containing the message descriptor data, and message to be sent. The message descriptor can be altered as a consequence of this method. The values in the message descriptor immediately after the completion of this method are the values that were put to the queue or published to the topic.

The following reason codes are returned to a reconnectable client:

- `MQRC_CALL_INTERRUPTED` if the connection is broken while running a Put call on a persistent message and the reconnection is successful.
- `MQRC_NONE` if the connection is successful while running a Put call on a non-persistent message (see [Application Recovery](#)).

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of `MQPutMessageOptions` is used.

If you use the `MQPMO_LOGICAL_ORDER` option in a reconnectable client, the `MQRC_RECONNECT_INCOMPATIBLE` reason code is returned.

Note: For simplicity and performance, if you want to put a single message to a queue, use `MQQueueManager.Put` object. You should have an `MQQueue` object for this.

```
public void PutForwardMessage(MQMessage message);  
public void PutForwardMessage(MQMessage message, MQPutMessageOptions  
putMessageOptions);
```

Throws `MQException`

Put a message being forwarded onto the queue, where *message* is the original message.

message

An `MQMessage` object containing the message descriptor data, and message to be sent. The message descriptor can be altered as a consequence of this method. The values in the message descriptor immediately after the completion of this method are the values that were put to the queue or published to the topic.

The following reason codes are returned to a reconnectable client:

- `MQRC_CALL_INTERRUPTED` if the connection is broken while running a Put call on a persistent message and the reconnection is successful.
- `MQRC_NONE` if the connection is successful while running a Put call on a non-persistent message (see [Application Recovery](#)).

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of `MQPutMessageOptions` is used.

If you use the `MQPMO_LOGICAL_ORDER` option in a reconnectable client, the `MQRC_RECONNECT_INCOMPATIBLE` reason code is returned.

```
public void PutReplyMessage(MQMessage message)  
public void PutReplyMessage(MQMessage message, MQPutMessageOptions  
putMessageOptions)
```

Throws `MQException`.

Put a reply message onto the queue, where *message* is the original message.

message

Contains the message descriptor and the returned message data. Some of the fields in the message descriptor are input parameters. It is important to ensure that the MessageId and CorrelationId input parameters are set as required.

A reconnectable client returns the reason code MQRC_BACKED_OUT after successful reconnection, for messages received under MQGM_SYNCPOINT.

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of MQPutMessageOptions is used.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

```
public void PutReportMessage(MQMessage message)  
public void PutReportMessage(MQMessage message, MQPutMessageOptions  
putMessageOptions)
```

Throws MQException.

Put a report message onto the queue, where *message* is the original message.

message

Contains the message descriptor and the returned message data. Some of the fields in the message descriptor are input parameters. It is important to ensure that the MessageId and CorrelationId input parameters are set as required.

A reconnectable client returns the reason code MQRC_BACKED_OUT after successful reconnection, for messages received under MQGM_SYNCPOINT.

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of MQPutMessageOptions is used.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

Constructors

```
public MQQueue MQQueueManager.AccessQueue(string queueName, int openOptions);  
public MQQueue MQQueueManager.AccessQueue(string queueName, int openOptions,  
string queueManagerName, string dynamicQueueName, string alternateUserId);
```

Throws MQException.

Accesses a queue on this queue manager.

You can get or browse messages, put messages, inquire about the attributes of the queue or set the attributes of the queue. If the queue named is a model queue, a dynamic local queue is created. Query the name attribute of the resultant MQQueue object to find out the name of the dynamic queue.

queueName

Name of queue to open.

openOptions

Options that control the opening of the queue.

MQC.MQOO_ALTERNATE_USER_AUTHORITY

Validate with the specified user identifier.

MQC.MQOO_BIND_AS_QDEF

Use default binding for queue.

MQC.MQOO_BIND_NOT_FIXED

Do not bind to a specific destination.

MQC.MQOO_BIND_ON_OPEN

Bind handle to destination when queue is opened.

MQC.MQOO_BROWSE

Open to browse message.

MQC.MQOO_FAIL_IF QUIESCING

Fail if the queue manager is quiescing.

MQC.MQOO_INPUT_AS_Q_DEF

Open to get messages using queue-defined default.

MQC.MQOO_INPUT_SHARED

Open to get messages with shared access.

MQC.MQOO_INPUT_EXCLUSIVE

Open to get messages with exclusive access.

MQC.MQOO_INQUIRE

Open for inquiry - required if you want to query properties.

MQC.MQOO_OUTPUT

Open to put messages.

MQC.MQOO_PASS_ALL_CONTEXT

Allow all context to be passed.

MQC.MQOO_PASS_IDENTITY_CONTEXT

Allow identity context to be passed.

MQC.MQOO_SAVE_ALL_CONTEXT

Save context when message retrieved.

MQC.MQOO_SET

Open to set attributes - required if you want to set properties.

MQC.MQOO_SET_ALL_CONTEXT

Allows all context to be set.

MQC.MQOO_SET_IDENTITY_CONTEXT

Allows identity context to be set.

queueManagerName

Name of the queue manager on which the queue is defined. A name that is entirely blank or null denotes the queue manager to which the MQQueueManager object is connected.

dynamicQueueName

dynamicQueueName is ignored unless *queueName* specifies the name of a model queue. If it does, *dynamicQueueName* specifies the name of the dynamic queue to be created. A blank or null name is not valid if *queueName* specifies the name of a model queue. If the last nonblank character in the name is an asterisk, *, the queue manager replaces the asterisk with a string of characters. The characters guarantee that the name generated for the queue is unique on this queue manager.

alternateUserId

If MQC.MQOO_ALTERNATE_USER_AUTHORITY is specified in the *openOptions* parameter, *alternateUserId* specifies the alternate user identifier that is used to check the authorization for the open. If MQC.MQOO_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be left blank, or null.

```
public MQQueue(MQQueueManager queueManager, string queueName, int openOptions,  
string queueManagerName, string dynamicQueueName, string alternateUserId);
```

Throws MQException.

Accesses a queue on queueManager.

You can get or browse messages, put messages, inquire about the attributes of the queue or set the attributes of the queue. If the queue named is a model queue, a dynamic local queue is created. Query the name attribute of the resultant MQQueue object to find out the name of the dynamic queue.

queueManager

Queue manager to access queue on.

queueName

Name of queue to open.

openOptions

Options that control the opening of the queue.

MQC.MQOO_ALTERNATE_USER_AUTHORITY

Validate with the specified user identifier.

MQC.MQOO_BIND_AS_QDEF

Use default binding for queue.

MQC.MQOO_BIND_NOT_FIXED

Do not bind to a specific destination.

MQC.MQOO_BIND_ON_OPEN

Bind handle to destination when queue is opened.

MQC.MQOO_BROWSE

Open to browse message.

MQC.MQOO_FAIL_IF QUIESCING

Fail if the queue manager is quiescing.

MQC.MQOO_INPUT_AS_Q_DEF

Open to get messages using queue-defined default.

MQC.MQOO_INPUT_SHARED

Open to get messages with shared access.

MQC.MQOO_INPUT_EXCLUSIVE

Open to get messages with exclusive access.

MQC.MQOO_INQUIRE

Open for inquiry - required if you want to query properties.

MQC.MQOO_OUTPUT

Open to put messages.

MQC.MQOO_PASS_ALL_CONTEXT

Allow all context to be passed.

MQC.MQOO_PASS_IDENTITY_CONTEXT

Allow identity context to be passed.

MQC.MQOO_SAVE_ALL_CONTEXT

Save context when message retrieved.

MQC.MQOO_SET

Open to set attributes - required if you want to set properties.

MQC.MQOO_SET_ALL_CONTEXT

Allows all context to be set.

MQC.MQOO_SET_IDENTITY_CONTEXT

Allows identity context to be set.

queueManagerName

Name of the queue manager on which the queue is defined. A name that is entirely blank or null denotes the queue manager to which the MQQueueManager object is connected.

dynamicQueueName

dynamicQueueName is ignored unless *queueName* specifies the name of a model queue. If it does, *dynamicQueueName* specifies the name of the dynamic queue to be created. A blank or null name is not valid if *queueName* specifies the name of a model queue. If the last nonblank character in the name is an asterisk, *, the queue manager replaces the asterisk with a string of characters. The characters guarantee that the name generated for the queue is unique on this queue manager.

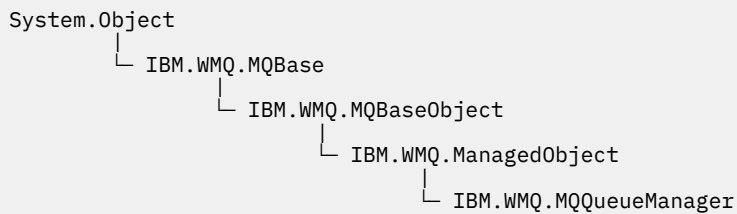
alternateUserId

If MQC.MQ00_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter, *alternateUserId* specifies the alternate user identifier that is used to check the authorization for the open. If MQC.MQ00_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be left blank, or null.

MQQueueManager.NET class

Use MQQueueManager to connect to a queue manager and access queue manager objects. It also controls transactions. The MQQueueManager constructor creates either a client or server connection.

Class



```
public class IBM.WMQ.MQQueueManager extends IBM.WMQ.MQManagedObject;
```

- [“Properties” on page 1798](#)
- [“Methods” on page 1801](#)
- [“Constructors” on page 1807](#)

Properties

Test for MQException being thrown when getting properties.

public int AccountingConnOverride {get;}

Whether applications can override the setting of the MQI accounting and queue accounting values.

public int AccountingInterval {get;}

How long before intermediate accounting records are written (in seconds).

public int ActivityRecording {get;}

Controls the generation of activity reports.

public int AdoptNewMCACheck {get;}

Specifies which elements are checked to determine whether the MCA is adopted when a new inbound channel is detected. To be adopted, the MCA name must match the name of an active MCA.

public int AdoptNewMCAInterval {get;}

The amount of time, in seconds, that the new channel waits for the orphaned channel to end.

public int AdoptNewMCAType {get;}

Whether an orphaned MCA instance is to be adopted (restarted) when a new inbound channel request is detected matching the AdoptNewMCACheck value.

public int BridgeEvent {get;}

Whether IMS bridge events are generated.

public int ChannelEvent {get;}

Whether channel events are generated.

public int ChannelInitiatorControl {get;}

Whether the channel initiator starts automatically when the queue manager starts.

public int ChannelInitiatorAdapters {get;}

The number of adapter subtasks to process IBM MQ calls.

public int ChannelInitiatorDispatchers {get;}

The number of dispatchers to use for the channel initiator.

public int ChannelInitiatorTraceAutoStart {get;}

Specifies whether the channel initiator trace starts automatically.

public int ChannelInitiatorTraceTableSize {get;}

The size, in megabytes, of the trace data space of a channel initiator.

public int ChannelMonitoring {get;}

Whether channel monitoring is used.

public int ChannelStatistics {get;}

Controls the collection of statistics data for channels.

public int CharacterSet {get;}

Returns the coded character set identifier (CCSID) of the queue manager. CharacterSet is used by the queue manager for all character string fields in the application programming interface.

public int ClusterSenderMonitoring {get;}

Controls the collection of online monitoring data for automatically defined cluster sender channels.

public int ClusterSenderStatistics {get;}

Controls the collection of statistics data for automatically defined cluster sender channels.

public int ClusterWorkLoadMRU {get;}

The maximum number of outbound cluster channels.

public int ClusterWorkLoadUseQ {get;}

The default value of the MQQueue property, ClusterWorkLoadUseQ, if it specifies a value of QMGR.

public int CommandEvent {get;}

Specifies whether command events are generated.

public string CommandInputQueueName {get;}

Returns the name of the command input queue defined on the queue manager. Applications can send commands to this queue, if authorized to do so.

public int CommandLevel {get;}

Indicates the function level of the queue manager. The set of functions that correspond to a particular function level depends on the platform. On a particular platform, you can rely on every queue manager supporting the functions at the lowest functional level common to all the queue managers.

public int CommandLevel {get;}

Whether the command server starts automatically when the queue manager starts.

public string DNSGroup {get;}

No longer used.

public int DNSWLM {get;}

No longer used.

public int IPAddressVersion {get;}

Which IP protocol (IPv4 or IPv6) to use for a channel connection.

public boolean IsConnected {get;}

Returns the value of the isConnected.

If true, a connection to the queue manager has been made, and is not known to be broken. Any calls to IsConnected do not actively attempt to reach the queue manager, so it is possible that physical connectivity can break, but IsConnected can still return true. The IsConnected state is only updated when activity, for example, putting a message, getting a message, is performed on the queue manager.

If false, a connection to the queue manager has not been made, or has been broken, or has been disconnected.

public int KeepAlive {get;}

Specifies whether the TCP KEEPALIVE facility is to be used to check that the other end of the connection is still available. If it is unavailable, the channel is closed.

public int ListenerTimer {get;}

The time interval, in seconds, between attempts by IBM MQ to restart the listener after an APPC or TCP/IP failure.

public int LoggerEvent {get;}

Whether logger events are generated.

public string LU62ARMSuffix {get;}

The suffix of the APPCPM member of SYS1.PARMLIB. This suffix nominates the LUADD for this channel initiator. When automatic restart manager (ARM) restarts the channel initiator, the z/OS command SET APPC=xx is issued.

public string LUGroupName {get; z/os}

The generic LU name to be used by the LU 6.2 listener that handles inbound transmissions for the queue-sharing group.

public string LUName {get;}

The name of the LU to use for outbound LU 6.2 transmissions.

public int MaximumActiveChannels {get;}

The maximum number of channels that can be active at any time.

public int MaximumCurrentChannels {get;}

The maximum number of channels that can be current at any time (including server-connection channels with connected clients).

public int MaximumLU62Channels {get;}

The maximum number of channels that can be current, or clients that can be connected, that use the LU 6.2 transmission protocol.

public int MaximumMessageLength {get;}

Returns the maximum length of a message (in bytes) that can be handled by the queue manager. No queue can be defined with a maximum message length greater than MaximumMessageLength.

public int MaximumPriority {get;}

Returns the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to this value. Throws MQException if you call this method after disconnecting from the queue manager.

public int MaximumTCPChannels {get;}

The maximum number of channels that can be current, or clients that can be connected, that use the TCP/IP transmission protocol.

public int MQIAccounting {get;}

Controls the collection of accounting information for MQI data.

public int MQIStatistics {get;}

Controls the collection of statistics monitoring information for the queue manager.

public int OutboundPortMax {get;}

The maximum value in the range of port numbers to be used when binding outgoing channels.

public int OutboundPortMin {get;}

The minimum value in the range of port numbers to be used when binding outgoing channels.

public int QueueAccounting {get;}

Whether class 3 accounting (thread-level and queue-level accounting) data is to be used for all queues.

public int QueueMonitoring {get;}

Controls the collection of online monitoring data for queues.

public int QueueStatistics {get;}

Controls the collection of statistics data for queues.

public int ReceiveTimeout {get;}

The length of time that a TCP/IP channel waits to receive data, including heartbeats, from its partner before returning to the inactive state.

public int ReceiveTimeoutMin {get;}

The minimum length of time that a TCP/IP channel waits to receive data, including heartbeats, from its partner before returning to an inactive state.

public int ReceiveTimeoutType {get;}

The qualifier to apply to the value in ReceiveTimeout.

public int SharedQueueQueueManagerName {get;}

Specifies how to deliver messages to a shared queue. If the put specifies a different queue manager from the same queue sharing group as the target queue manager, the message is delivered in two ways:

MQC.MQSQQM_USE

Messages are delivered to the object queue manager before being put on the shared queue.

MQCMQSQQM_IGNORE

Messages are put directly on the shared queue.

public int SSLEvent {get;}

Whether SSL events are generated.

public int SSLFips {get;}

Whether only FIPS-certified algorithms are to be used if cryptography is performed in IBM MQ, rather than cryptographic hardware.

public int SSLKeyResetCount {get;}

Indicates the number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated.

public int ClusterSenderStatistics {get;}

Specifies the interval, in minutes, between consecutive gatherings of statistics.

public int SyncpointAvailability {get;}

Indicates whether the queue manager supports units of work and sync points with the MQQueue.get and MQQueue.put methods.

public string TCPName {get;}

The name of either the only, or default, TCP/IP system to be used, depending on the value of TCPStackType.

public int TCPStackType {get;}

Specifies whether the channel initiator uses only the TCP/IP address space specified in TCPName. Alternatively, the channel initiator can bind to any TCP/IP address.

public int TraceRouteRecording {get;}

Controls the recording of route tracing information.

Methods

public MQProcess AccessProcess(string processName, int openOptions);

public MQProcess AccessProcess(string processName, int openOptions, string queueManagerName, string alternateUserId);

Throws MQException.

Access an IBM MQ process on this queue manager to inquire on process attributes.

processName

The name of the process to open.

openOptions

Options that control the opening of the process. The valid options that can be added, or combined using a bitwise OR, are:

- MQC.MQ00_FAIL_IF QUIESCING
- MQC.MQ00_INQUIRE
- MQC.MQ00_SET

- MQC.MQOO_ALTERNATE_USER_AUTHORITY

queueManagerName

The name of the queue manager on which the process is defined. You can leave a blank or null queue manager name if the queue manager is the same as the one the process is accessing.

alternateUserId

If MQC.MQOO_ALTERNATE_USER_AUTHORITY is specified in the *openOptions* parameter, *alternateUserId* specifies the alternative user ID used to check the authorization for the action. If MQOO_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be blank or null.

Default user authority is used for connection to the queue manager if MQC.MQOO_ALTERNATE_USER_AUTHORITY is not specified.

```
public MQQueue AccessQueue(string queueName, int openOptions);  
public MQQueue AccessQueue(string queueName, int openOptions, string  
queueManagerName, string dynamicQueueName, string alternateUserId);
```

Throws MQException.

Accesses a queue on this queue manager.

You can get or browse messages, put messages, inquire about the attributes of the queue or set the attributes of the queue. If the queue named is a model queue, a dynamic local queue is created. Query the name attribute of the resultant MQQueue object to find out the name of the dynamic queue.

queueName

Name of queue to open.

openOptions

Options that control the opening of the queue.

MQC.MQOO_ALTERNATE_USER_AUTHORITY

Validate with the specified user identifier.

MQC.MQOO_BIND_AS_QDEF

Use default binding for queue.

MQC.MQOO_BIND_NOT_FIXED

Do not bind to a specific destination.

MQC.MQOO_BIND_ON_OPEN

Bind handle to destination when queue is opened.

MQC.MQOO_BROWSE

Open to browse message.

MQC.MQOO_FAIL_IF QUIESCING

Fail if the queue manager is quiescing.

MQC.MQOO_INPUT_AS_Q_DEF

Open to get messages using queue-defined default.

MQC.MQOO_INPUT_SHARED

Open to get messages with shared access.

MQC.MQOO_INPUT_EXCLUSIVE

Open to get messages with exclusive access.

MQC.MQOO_INQUIRE

Open for inquiry - required if you want to query properties.

MQC.MQOO_OUTPUT

Open to put messages.

MQC.MQOO_PASS_ALL_CONTEXT

Allow all context to be passed.

MQC.MQOO_PASS_IDENTITY_CONTEXT

Allow identity context to be passed.

MQC.MQOO_SAVE_ALL_CONTEXT

Save context when message retrieved.

MQC.MQOO_SET

Open to set attributes - required if you want to set properties.

MQC.MQOO_SET_ALL_CONTEXT

Allows all context to be set.

MQC.MQOO_SET_IDENTITY_CONTEXT

Allows identity context to be set.

queueManagerName

Name of the queue manager on which the queue is defined. A name that is entirely blank or null denotes the queue manager to which the MQQueueManager object is connected.

dynamicQueueName

dynamicQueueName is ignored unless *queueName* specifies the name of a model queue. If it does, *dynamicQueueName* specifies the name of the dynamic queue to be created. A blank or null name is not valid if *queueName* specifies the name of a model queue. If the last nonblank character in the name is an asterisk, *, the queue manager replaces the asterisk with a string of characters. The characters guarantee that the name generated for the queue is unique on this queue manager.

alternateUserId

If MQC.MQOO_ALTERNATE_USER_AUTHORITY is specified in the openOptions parameter, *alternateUserId* specifies the alternate user identifier that is used to check the authorization for the open. If MQC.MQOO_ALTERNATE_USER_AUTHORITY is not specified, *alternateUserId* can be left blank, or null.

```
public MQTopic AccessTopic( MQDestination destination, string topicName, string
topicObject, int options);
public MQTopic AccessTopic( MQDestination destination, string topicName, string
topicObject, int options, string alternateUserId);
public MQTopic AccessTopic( MQDestination destination, string topicName, string
topicObject, int options, string alternateUserId, string subscriptionName);
public MQTopic AccessTopic( MQDestination destination, string topicName, string
topicObject, int options, string alternateUserId, string subscriptionName,
System.Collections.Hashtable properties);
public MQTopic AccessTopic(string topicName, string topicObject, int openAs,
int options);
public MQTopic AccessTopic(string topicName, string topicObject, int openAs,
int options, string alternateUserId);
public MQTopic AccessTopic(string topicName, string topicObject, int options,
string alternateUserId, string subscriptionName);
public MQTopic AccessTopic(string topicName, string topicObject, int options,
string alternateUserId, string subscriptionName, System.Collections.Hashtable
properties);
```

Access a topic on this queue manager.

MQTopic objects are closely related to administrative topic objects, which are sometimes called topic objects. On input, *topicObject* points to an administrative topic object. The MQTopic constructor obtains a topic string from the topic object and combines it with *topicName* to create a topic name. Either or both *topicObject* or *topicName* can be null. The topic name is matched to the topic tree, and the name of the closest matching administrative topic object is returned in *topicObject*.

The topics that are associated with the MQTopic object are the result of combining two topic strings. The first topic string is defined by the administrative topic object identified by *topicObject*. The second topic string is *topicString*. The resulting topic string associated with the MQTopic object can identify multiple topics by including wild cards.

Depending on whether the topic is opened for publishing or subscribing, you can use the `MQTopic.Put` methods to publish on topics, or `MQTopic.Get` methods to receive publications on topics. If you want to publish and subscribe to the same topic, you must access the topic twice, once for publish and once for subscribe.

If you create an `MQTopic` object for subscription, without providing an `MQDestination` object, a managed subscription is assumed. If you pass a queue as an `MQDestination` object, an unmanaged subscription is assumed. You must ensure the subscription options you set are consistent with the subscription being managed or unmanaged.

destination

destination is an `MQQueue` instance. By providing *destination*, `MQTopic` is opened as an unmanaged subscription. Publications on the topic are delivered to the queue accessed as *destination*.

topicName

A topic string that is the second part of the topic name. *topicName* is concatenated with the topic string defined in the *topicObject* administrative topic object. You can set *topicName* to null, in which case the topic name is defined by the topic string in *topicObject*.

topicObject

On input, *topicObject* is the name of the topic object that contains the topic string that forms the first part of the topic name. The topic string in *topicObject* is concatenated with *topicName*. The rules for constructing topic strings are defined in [Combining topic strings](#).

On output, *topicObject* contains the name of the administrative topic object that is the closest match in the topic tree to the topic identified by the topic string.

openAs

Access the topic to publish or subscribe. The parameter can contain only one of these options:

- `MQC.MQTOPIC_OPEN_AS_SUBSCRIPTION`
- `MQC.MQTOPIC_OPEN_AS_PUBLICATION`

options

Combine the options that control the opening of the topic for either publication or subscription. Use `MQC.MQSO_*` constants to access a topic for subscription and `MQC.MQOO_*` constants to access a topic for publication.

If more than one option is required, add the values together, or combine the option values using the bitwise OR operator.

alternateUserId

Specify the alternate user ID that is used to check for the required authorization to finish the operation. You must specify *alternateUserId*, if either `MQC.MQOO_ALTERNATE_USER_AUTHORITY` or `MQC.MQSO_ALTERNATE_USER_AUTHORITY` is set in the options parameter.

subscriptionName

subscriptionName is required if the options `MQC.MQSO_DURABLE` or `MQC.MQSO_ALTER` are provided. In both cases, `MQTopic` is implicitly opened for subscription. An exception is thrown if the `MQC.MQSO_DURABLE` is set, and the subscription exists, or if `MQC.MQSO_ALTER` is set, and the subscription does not exist.

properties

Set any of the special subscription properties listed using a hash table. Specified entries in the hash table are updated with output values. Entries are not added to the hash table to report output values.

- `MQC.MQSUB_PROP_ALTERNATE_SECURITY_ID`
- `MQC.MQSUB_PROP_SUBSCRIPTION_EXPIRY`
- `MQC.MQSUB_PROP_SUBSCRIPTION_USER_DATA`
- `MQC.MQSUB_PROP_SUBSCRIPTION_CORRELATION_ID`

- MQC.MQSUB_PROP_PUBLICATION_PRIORITY
- MQC.MQSUB_PROP_PUBLICATION_ACCOUNTING_TOKEN
- MQC.MQSUB_PROP_PUBLICATION_APPLICATIONID_DATA

public MQAsyncStatus GetAsyncStatus();

Throws MQException

Returns an MQAsyncStatus object, which represents the asynchronous activity for the queue manager connection.

public void Backout();

Throws MQException.

Backout any messages that were read or written within sync point since the last sync point.

Messages that were written with the MQC.MQPMO_SYNCPOINT flag set are removed from queues. Messages read with the MQC.MQGMO_SYNCPOINT flag are reinstated on the queues they came from. If the messages are persistent, the changes are logged.

For reconnectable clients, the MQRC_NONE reason code is returned to a client after reconnection is successful.

public void Begin();

Throws MQException.

Begin is supported only in server bindings mode. It starts a global unit of work.

public void Commit();

Throws MQException.

Commit any messages that were read or written within sync point since the last sync point.

Messages written with the MQC.MQPMO_SYNCPOINT flag set are made available to other applications. Messages retrieved with the MQC.MQGMO_SYNCPOINT flag set are deleted. If the messages are persistent, the changes are logged.

The following reason codes are returned to a reconnectable client:

- MQRC_CALL_INTERRUPTED if connection is lost while carrying out the commit call.
- MQRC_BACKED_OUT if the commit call is issued after reconnection.

Disconnect();

Throws MQException.

Close the connection to the queue manager. All objects accessed on this queue manager are not longer accessible to this application. To reaccess the objects, create a MQQueueManager object.

Generally, any work performed as part of a unit of work is committed. However, if the unit of work is managed by .NET, the unit of work might be rolled back.

```

public void Put(int type, string destinationName, MQMessage message);
public void Put(int type, string destinationName, MQMessage message
MQPutMessageOptions putMessageOptions);
public void Put(int type, string destinationName, string queueManagerName,
string topicString, MQMessage message);
public void Put(string queueName, MQMessage message);
public void Put(string queueName, MQMessage message, MQPutMessageOptions
putMessageOptions);
public void Put(string queueName, string queueManagerName, MQMessage message);
public void Put(string queueName, string queueManagerName, MQMessage message,
MQPutMessageOptions putMessageOptions);
public void Put(string queueName, string queueManagerName, MQMessage message,
MQPutMessageOptions putMessageOptions, string alternateUserId);

```

Throws MQException.

Places a single message onto a queue or topic without creating an MQQueue or MQTopic object first.

queueName

The name of the queue onto which to place the message.

destinationName

The name of a destination object. It is either a queue or a topic depending on the value of *type*.

type

The type of destination object. You must not combine the options.

MQC.MQOT_Q

Queue

MQC.MQOT_TOPIC

Topic

queueManagerName

The name of the queue manager or queue manager alias, on which the queue is defined. If type MQC.MQOT_TOPIC is specified this parameter is ignored.

If the queue is a model queue, and the resolved queue manager name is not this queue manager, an MQException is thrown.

topicString

topicString is combined with the topic name in the *destinationName* topic object.

topicString is ignored if *destinationName* is a queue.

message

The message to send. Message is an input/output object.

The following reason codes are returned to a reconnectable client:

- MQRC_CALL_INTERRUPTED if the connection is broken while performing a Put call on a persistent message.
- MQRC_NONE if the connection is successful while performing a Put call on a non-persistent message (see [Application Recovery](#)).

putMessageOptions

Options controlling the actions of the put.

If you omit *putMessageOptions*, a default instance of *putMessageOptions* is created. *putMessageOptions* is an input/output object.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

alternateUserId

Specifies an alternate user identifier used to check authorization when placing the message on a queue.

You can omit *alternateUserId* if you do not set `MQC.MQOO_ALTERNATE_USER_AUTHORITY` in *putMessageOptions*. If you set `MQC.MQOO_ALTERNATE_USER_AUTHORITY`, you must also set *alternateUserId*. *alternateUserId* has not effect unless you also set `MQC.MQOO_ALTERNATE_USER_AUTHORITY`.

Constructors

```
public MQQueueManager();  
public MQQueueManager(string queueManagerName);  
public MQQueueManager(string queueManagerName, Int options);  
public MQQueueManager(string queueManagerName, Int options, string channel,  
string connName);  
public MQQueueManager(string queueManagerName, string channel, string  
connName);  
public MQQueueManager(string queueManagerName, System.Collections.Hashtable  
properties);
```

Throws MQException.

Creates a connection to a queue manager. Select between creating a client connection or a server connection.

You must have inquire (inq) authority on the queue manager when attempting to connect to the queue manager. Without inquire authority, the connection attempt fails.

A client connection is created if one of the following conditions is true:

1. *channel* or *connName* are specified in the constructor.
2. *HostName*, *Port*, or *Channel* are specified in *properties*.
3. *MQEnvironment.HostName*, *MQEnvironment.Port*, or *MQEnvironment.Channel* are specified.

The values of the connection properties are defaulted in the order shown. The *channel* and *connName* in the constructor take precedence over the property values in the constructor. The constructor property values take precedence of the *MQEnvironment* properties.

The host name, channel name, and port are defined in the *MQEnvironment* class.

queueManagerName

Name of the queue manager, or queue manager group to connect to.

Omit the parameter, or leave it null, or blank to make a default queue manager selection. The default queue manager connection on a server is to the default queue manager on the server. The default queue manager connection on a client connection is to the queue manager the listener is connected to.

options

Specify MQCNO connection options. The values must be applicable to the type of connection being made. For example, if you specify the following server connection properties for a client connection an MQException is thrown.

- `MQC.MQCNO_FASTPATH_BINDING`
- `MQC.MQCNO_STANDARD_BINDING`

properties

The properties parameter takes a series of key/value pairs that override the properties set by *MQEnvironment*; see the example, [“Override MQEnvironment properties”](#) on page 1810. The following properties can be overridden:

- MQC.CONNECT_OPTIONS_PROPERTY
- MQC.CONNECTION_NAME_PROPERTY
- MQC.ENCRYPTION_POLICY_SUITE_B
- MQC.HOST_NAME_PROPERTY
- MQC.PORT_PROPERTY
- MQC.CHANNEL_PROPERTY
- MQC.SSL_CIPHER_SPEC_PROPERTY
- MQC.SSL_PEER_NAME_PROPERTY
- MQC.SSL_CERT_STORE_PROPERTY
- MQC.SSL_CRYPTOHARDWARE_PROPERTY
- MQC.SECURITY_EXIT_PROPERTY
- MQC.SECURITY_USERDATA_PROPERTY
- MQC.SEND_EXIT_PROPERTY
- MQC.SEND_USERDATA_PROPERTY
- MQC.RECEIVE_EXIT_PROPERTY
- MQC.RECEIVE_USERDATA_PROPERTY
- MQC.USER_ID_PROPERTY
- MQC.PASSWORD_PROPERTY
- MQC.MQAIR_ARRAY
- MQC.KEY_RESET_COUNT
- MQC.FIPS_REQUIRED
- MQC.HDR_CMP_LIST
- MQC.MSG_CMP_LIST
- MQC.TRANSPORT_PROPERTY

channel

Name of a server connection channel

connName

Connection name in the format *HostName (Port)* .

You can supply a list of *hostnames* and *ports* as an argument to the constructor `MQQueueManager (String queueManagerName, Hashtable properties)` using `CONNECTION_NAME_PROPERTY`.

For example:

```
ConnectionName = "fred.mq.com(2344),nick.mq.com(3746),tom.mq.com(4288)";
Hashtable Properties=new Hashtable();
properties.Add(MQC.CONNECTION_NAME_PROPERTY,ConnectionName);
MQQueueManager qmgr=new MQQueue Manager("qmgrname",properties);
```

When a connection attempt is made, the connection name list is processed in order. If the connection attempt to the first host name and port fails, then connection to the second pair of attributes is attempted. The client repeats this process until either a successful connection is made or the list is exhausted. If the list is exhausted, an appropriate reason code and completion code is returned to the client application.

When a port number is not provided for the connection name, the default port (configured in `mqclient.ini`) is used.

Set the Connection List

You can set the connection list by using the following methods when the automatic client reconnection options are set:

Set the connection list through MQSERVER

You can set the connection list through the command prompt.

At the command prompt, set the following command:

```
MQSERVER=SYSTEM.DEF.SVRCONN/TCP/Hostname1(Port1),Hostname2(Por2),Hostname3(Port3)
```

For example:

```
MQSERVER=SYSTEM.DEF.SVRCONN/TCP/fred.mq.com(5266),nick.mq.com(6566),jack.mq.com(8413)
```

If you set the connection in the MQSERVER, do not set it in the application.

If you set the connection list in the application, the application overwrites whatever is set in the MQSERVER environment variable.

Set the connection list through the application

You can set the connection list in the application by specifying the host name and port properties.

```
String connName = "fred.mq.com(2344), nick.mq.com(3746), chris.mq.com(4288)";  
MQQueueManager qm = new MQQueueManager("QM1", "TestChannel", connName);
```

Set the connection list through app.config

App.config is an XML file where you specify the key-value pairs.

In the connection list specify

```
<app.Settings>  
<add key="Connection1" value="Hostname1(Port1)"/>  
<add key="Connection2" value="Hostname2(Port2)"/>  
</app.Settings>
```

For example:

```
<app.Settings>  
<add key="Connection1" value="fred.mq.com(2966)"/>  
<add key="Connection2" value="alex.mq.com(6533)"/>  
</app.Settings>
```

You can directly change the connection list in the app.config file.

Set the connection list through MQEnvironment

To set the Connection list through the MQEnvironment, use the *ConnectionName* property.

```
MQEnvironment.ConnectionName = "fred.mq.com(4288),alex.mq.com(5211);
```

The *ConnectionName* property overwrites the host name and port properties set in the MQEnvironment.

Create a client connection

The following example shows you how to create a client connection to a queue manager. You can create a client connection by setting the MQEnvironment variables before creating a new MQQueueManager Object.

```

MQEnvironment.Hostname = "fred.mq.com"; // host to connect to
MQEnvironment.Port      = 1414;         // port to connect to
                                     // If not explicitly set,
                                     // defaults to 1414
                                     // (the default WebSphere MQ port)
MQEnvironment.Channel   = "channel.name"; // the case sensitive
                                     // name of the
                                     // SVR CONN channel on
                                     // the queue manager
MQQueueManager qMgr     = new MQQueueManager("MYQM");

```

Figure 43. Client connection

Override MQEnvironment properties

The following example shows you how to create a queue manager with its user ID and password defined in a hash table.

```

Hashtable properties = new Hashtable();
properties.Add( MQC.USER_ID_PROPERTY, "ExampleUserId" );
properties.Add( MQC.PASSWORD_PROPERTY, "ExamplePassword" );

try
{
    MQQueueManager qMgr = new MQQueueManager("qmgrname", properties);
}
catch (MQException mqe)
{
    System.Console.WriteLine("Connect failed with " + mqe.Message);
    return((int)mqe.Reason);
}

```

Figure 44. Overriding MQEnvironment properties

Create a reconnectable connection

The following example shows you how to automatically reconnect a client to a Queue Manager.

```

Hashtable properties = new Hashtable(); // The queue manager name and the
                                     // properties how it has to be connected

properties.Add(MQC.CONNECT_OPTIONS_PROPERTY, MQC.MQCNO_RECONNECT); // Options
                                     // through which reconnection happens

properties.Add(MQC.CONNECTION_NAME_PROPERTY, "fred.mq.com(4789),nick.mq.com(4790)"); // The list
                                     // of queue managers through which reconnection happens

MQ QueueManager qmgr = new MQQueueManager("qmgrname", properties);

```

Figure 45. Automatically reconnecting a client to a queue manager

MQSubscription.NET class

Use MQSubscription to request that retained publications are sent to the subscriber. MQSubscription is a property of an MQTopic object opened for subscription.

Class

```

System.Object
├── IBM.WMQ.MQBase
│   └── IBM.WMQ.MQBaseObject
│       └── IBM.WMQ.MQManagedObject
│           └── IBM.WMQ.MQSubscription

```

```
public class IBM.WMQ.MQSubscription extends IBM.WMQ.MQManagedObject;
```

- [“Properties” on page 1811](#)
- [“Methods” on page 1811](#)
- [“Constructors” on page 1811](#)

Properties

Access subscription properties using the `MQManagedObject` class; see [“Properties” on page 1771](#).

Methods

Access subscription Inquire, Set and Get methods using the `MQManagedObject` class; see [“Methods” on page 1771](#).

public int RequestPublicationUpdate(int options);

Throws `MQException`.

Request an updated publication for the current topic. If the queue manager has a retained publications for the topic, they are sent to the subscriber.

Before calling `RequestPublicationUpdate`, open a topic for subscription to obtain an `MQSubscription` object.

Typically, open the subscription with the `MQC.MQSO_PUBLICATIONS_ON_REQUEST` option. If no wildcards are present in the topic string, then only one publication is sent as a result of this call. If the topic string contains wildcards, many publications might be sent. The method returns the number of retained publications that are sent to the subscription queue. There is no guarantee that this many publications are received, especially if they are non-persistent messages.

options

MQC.MQSRO_FAIL_IF QUIESCING

The method fails if the queue manager is in a quiescent state. On z/OS, for a CICS or IMS application, `MQC.MQSRO_FAIL_IF QUIESCING` also forces the method to fail if the connection is in a quiescent state.

MQC.MQSRO_NONE

No options are specified.

Constructors

No Public constructor.

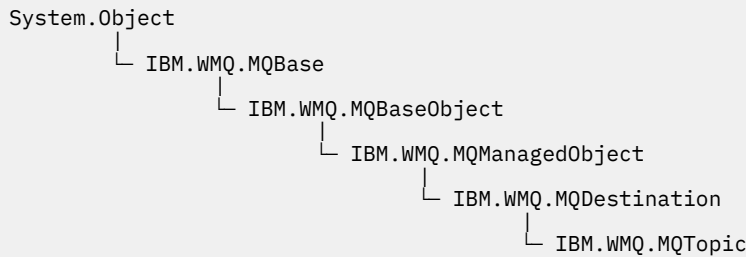
An `MQSubscription` object is returned in the `SubscriptionReference` property of an `MQTopic` object that is opened for subscription,

Call the `RequestPublicationUpdate` method. `MQSubscription` is a subclass of `MQManagedObject`. Use the reference to access the properties and methods of `MQManagedObject`.

MQTopic.NET class

Use `MQTopic` to publish or subscribe messages on a topic, or to query or set attributes of a topic. Create an `MQTopic` object for publishing or subscribing by using a constructor or the `MQQueueManager.AccessTopic` method.

Class



```
public class IBM.WMQ.MQTopic extends IBM.WMQ.MQDestination;
```

- [“Properties” on page 1812](#)
- [“Methods” on page 1812](#)
- [“Constructors” on page 1814](#)

Properties

Test for `MQException` being thrown when getting properties.

public Boolean IsDurable {get;}

Read only property that returns `True` if the subscription is durable or `False` otherwise. If the topic was opened for publication, the property is ignored and would always return `False`.

public Boolean IsManaged {get;};

Read only property that returns `True` if the subscription is managed by the queue manager, or `False` otherwise. If the topic was opened for publication, the property is ignored and would always return `False`.

public Boolean IsSubscribed {get;};

Read only property that returns `True` if the topic was opened for subscription and `False` if the topic was opened for publication.

public MQSubscription SubscriptionReference {get;};

Read only property that returns the `MQSubscription` object associated with a topic object opened for subscription. The reference is available if you want to modify the close options or start any of the objects methods.

public MQDestination UnmanagedDestinationReference {get;};

Read only property that returns the `MQQueue` associated with an unmanaged subscription. It is the destination specified when the topic object was created. The property returns null for any topic objects opened for publication or with a managed subscription.

Methods

public void Put(MQMessage message);

public void Put(MQMessage message, MQPutMessageOptions putMessageOptions);

Throws `MQException`.

Publishes a message to the topic.

Modifications to the `MQMessage` object after the `Put` call has been accomplished do not affect the actual message on the IBM MQ queue or publication topic.

`Put` updates the `MessageId` and `CorrelationId` properties of the `MQMessage` object and does not clear message data. Further `Put` or `Get` calls refer to the updated information in the `MQMessage` object. For example, in the following code snippet, the first message contains a and the second ab.

```
msg.WriteString("a");
q.Put(msg,pmo);
msg.WriteString("b");
q.Put(msg,pmo);
```

message

An MQMessage object containing the message descriptor data, and message to be sent. The message descriptor can be altered as a consequence of this method. The values in the message descriptor immediately after the completion of this method are the values that were put to the queue or published to the topic.

The following reason codes are returned to a reconnectable client:

- MQRC_CALL_INTERRUPTED if the connection is broken while running a Put call on a persistent message and the reconnection is successful.
- MQRC_NONE if the connection is successful while running a Put call on a non-persistent message (see [Application Recovery](#)).

putMessageOptions

Options controlling the action of the put.

If *putMessageOptions* is not specified the default instance of MQPutMessageOptions is used.

If you use the MQPMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

Note: For simplicity and performance, if you want to put a single message to a queue, use MQQueueManager.Put object. You should have an MQQueue object for this.

```
public void Get(MQMessage message);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions);  
public void Get(MQMessage message, MQGetMessageOptions getMessageOptions, int  
MaxMsgSize);
```

Throws MQException.

Retrieves a message from the topic.

This method uses a default instance of MQGetMessageOptions to do the get. The message option used is MQGMO_NOWAIT.

If the get fails, the MQMessage object is unchanged. If it succeeds, the message descriptor and message data portions of the MQMessage are replaced with the message descriptor and message data from the incoming message.

All calls to IBM MQ from a particular MQQueueManager are synchronous. Therefore, if you perform a get with wait, all other threads using the same MQQueueManager are blocked from making further IBM MQ calls until the Get call is accomplished. If you need multiple threads to access IBM MQ simultaneously, each thread must create its own MQQueueManager object.

message

Contains the message descriptor and the returned message data. Some of the fields in the message descriptor are input parameters. It is important to ensure that the MessageId and CorrelationId input parameters are set as required.

A reconnectable client returns the reason code MQRC_BACKED_OUT after successful reconnection, for messages received under MQGM_SYNCPOINT.

getMessageOptions

Options controlling the action of the get.

Using option MQC.MQGMO_CONVERT might result in an exception with reason code MQC.MQRC_CONVERTED_STRING_TOO_BIG when converting from single-byte character codes to double byte codes. In this case, the message is copied into the buffer without conversion.

If *getMessageOptions* is not specified, the message option used is MQGMO_NOWAIT.

If you use the MQGMO_LOGICAL_ORDER option in a reconnectable client, the MQRC_RECONNECT_INCOMPATIBLE reason code is returned.

MaxMsgSize

The largest message this message object is to receive. If the message on the queue is larger than this size, one of two things occurs:

- If the MQGMO_ACCEPT_TRUNCATED_MSG flag is set in the MQGetMessageOptions object, the message is filled with as much of the message data as possible. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_ACCEPTED reason code.
- If the MQGMO_ACCEPT_TRUNCATED_MSG flag is not set, the message remains on the queue. An exception is thrown with the MQCC_WARNING completion code and MQRC_TRUNCATED_MSG_FAILED reason code.

If *MaxMsgSize* is not specified, the whole message is retrieved.

Constructors

```
public MQTopic(MQQueueManager queueManager, MQDestination destination, string
topicName, string topicObject, int options);
public MQTopic(MQQueueManager queueManager, MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId);
public MQTopic(MQQueueManager queueManager, MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId, string
subscriptionName);
public MQTopic(MQQueueManager queueManager, MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId, string
subscriptionName, System.Collections.Hashtable properties);
public MQTopic(MQQueueManager queueManager, string topicName, string
topicObject, int openAs, int options);
public MQTopic(MQQueueManager queueManager, string topicName, string
topicObject, int openAs, int options, string alternateUserId);
public MQTopic(MQQueueManager queueManager, string topicName, string
topicObject, int options, string alternateUserId, string subscriptionName);
public MQTopic(MQQueueManager queueManager, string topicName, string
topicObject, int options, string alternateUserId, string subscriptionName,
System.Collections.Hashtable properties);
```

Access a topic on *queueManager*.

MQTopic objects are closely related to administrative topic objects, which are sometimes called topic objects. On input, *topicObject* points to an administrative topic object. The MQTopic constructor obtains a topic string from the topic object and combines it with *topicName* to create a topic name. Either or both *topicObject* or *topicName* can be null. The topic name is matched to the topic tree, and the name of the closest matching administrative topic object is returned in *topicObject*.

The topics that are associated with the MQTopic object are the result of combining two topic strings. The first topic string is defined by the administrative topic object identified by *topicObject*. The second topic string is *topicString*. The resulting topic string associated with the MQTopic object can identify multiple topics by including wild cards.

Depending on whether the topic is opened for publishing or subscribing, you can use the MQTopic.Put methods to publish on topics, or MQTopic.Get methods to receive publications on topics. If you want to publish and subscribe to the same topic, you must access the topic twice, once for publish and once for subscribe.

If you create an MQTopic object for subscription, without providing an MQDestination object, a managed subscription is assumed. If you pass a queue as an MQDestination object, an unmanaged subscription is assumed. You must ensure the subscription options you set are consistent with the subscription being managed or unmanaged.

queueManager

Queue manager to access a topic on.

destination

destination is an MQQueue instance. By providing *destination*, MQTopic is opened as an unmanaged subscription. Publications on the topic are delivered to the queue accessed as *destination*.

topicName

A topic string that is the second part of the topic name. *topicName* is concatenated with the topic string defined in the *topicObject* administrative topic object. You can set *topicName* to null, in which case the topic name is defined by the topic string in *topicObject*.

topicObject

On input, *topicObject* is the name of the topic object that contains the topic string that forms the first part of the topic name. The topic string in *topicObject* is concatenated with *topicName*. The rules for constructing topic strings are defined in [Combining topic strings](#).

On output, *topicObject* contains the name of the administrative topic object that is the closest match in the topic tree to the topic identified by the topic string.

openAs

Access the topic to publish or subscribe. The parameter can contain only one of these options:

- MQC.MQTOPIC_OPEN_AS_SUBSCRIPTION
- MQC.MQTOPIC_OPEN_AS_PUBLICATION

options

Combine the options that control the opening of the topic for either publication or subscription. Use MQC.MQSO_* constants to access a topic for subscription and MQC.MQOO_* constants to access a topic for publication.

If more than one option is required, add the values together, or combine the option values using the bitwise OR operator.

alternateUserId

Specify the alternate user ID that is used to check for the required authorization to finish the operation. You must specify *alternateUserId*, if either MQC.MQOO_ALTERNATE_USER_AUTHORITY or MQC.MQSO_ALTERNATE_USER_AUTHORITY is set in the options parameter.

subscriptionName

subscriptionName is required if the options MQC.MQSO_DURABLE or MQC.MQSO_ALTER are provided. In both cases, MQTopic is implicitly opened for subscription. An exception is thrown if the MQC.MQSO_DURABLE is set, and the subscription exists, or if MQC.MQSO_ALTER is set, and the subscription does not exist.

properties

Set any of the special subscription properties listed using a hash table. Specified entries in the hash table are updated with output values. Entries are not added to the hash table to report output values.

- MQC.MQSUB_PROP_ALTERNATE_SECURITY_ID
- MQC.MQSUB_PROP_SUBSCRIPTION_EXPIRY
- MQC.MQSUB_PROP_SUBSCRIPTION_USER_DATA
- MQC.MQSUB_PROP_SUBSCRIPTION_CORRELATION_ID
- MQC.MQSUB_PROP_PUBLICATION_PRIORITY
- MQC.MQSUB_PROP_PUBLICATION_ACCOUNTING_TOKEN
- MQC.MQSUB_PROP_PUBLICATION_APPLICATIONID_DATA

```

public MQTopic MQQueueManager.AccessTopic(MQDestination destination, string
topicName, string topicObject, int options);
public MQTopic MQQueueManager.AccessTopic(MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId);
public MQTopic MQQueueManager.AccessTopic(MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId, string
subscriptionName);
public MQTopic MQQueueManager.AccessTopic(MQDestination destination, string
topicName, string topicObject, int options, string alternateUserId, string
subscriptionName, System.Collections.Hashtable properties);
public MQTopic MQQueueManager.AccessTopic(string topicName, string topicObject,
int openAs, int options);
public MQTopic MQQueueManager.AccessTopic(string topicName, string topicObject,
int openAs, int options, string alternateUserId);
public MQTopic MQQueueManager.AccessTopic(string topicName, string topicObject,
int options, string alternateUserId, string subscriptionName);
public MQTopic MQQueueManager.AccessTopic(string topicName, string
topicObject, int options, string alternateUserId, string subscriptionName,
System.Collections.Hashtable properties);

```

Access a topic on this queue manager.

MQTopic objects are closely related to administrative topic objects, which are sometimes called topic objects. On input, *topicObject* points to an administrative topic object. The MQTopic constructor obtains a topic string from the topic object and combines it with *topicName* to create a topic name. Either or both *topicObject* or *topicName* can be null. The topic name is matched to the topic tree, and the name of the closest matching administrative topic object is returned in *topicObject*.

The topics that are associated with the MQTopic object are the result of combining two topic strings. The first topic string is defined by the administrative topic object identified by *topicObject*. The second topic string is *topicString*. The resulting topic string associated with the MQTopic object can identify multiple topics by including wild cards.

Depending on whether the topic is opened for publishing or subscribing, you can use the MQTopic .Put methods to publish on topics, or MQTopic .Get methods to receive publications on topics. If you want to publish and subscribe to the same topic, you must access the topic twice, once for publish and once for subscribe.

If you create an MQTopic object for subscription, without providing an MQDestination object, a managed subscription is assumed. If you pass a queue as an MQDestination object, an unmanaged subscription is assumed. You must ensure the subscription options you set are consistent with the subscription being managed or unmanaged.

destination

destination is an MQQueue instance. By providing *destination*, MQTopic is opened as an unmanaged subscription. Publications on the topic are delivered to the queue accessed as *destination*.

topicName

A topic string that is the second part of the topic name. *topicName* is concatenated with the topic string defined in the *topicObject* administrative topic object. You can set *topicName* to null, in which case the topic name is defined by the topic string in *topicObject*.

topicObject

On input, *topicObject* is the name of the topic object that contains the topic string that forms the first part of the topic name. The topic string in *topicObject* is concatenated with *topicName*. The rules for constructing topic strings are defined in [Combining topic strings](#).

On output, *topicObject* contains the name of the administrative topic object that is the closest match in the topic tree to the topic identified by the topic string.

openAs

Access the topic to publish or subscribe. The parameter can contain only one of these options:

- MQC.MQTOPIC_OPEN_AS_SUBSCRIPTION
- MQC.MQTOPIC_OPEN_AS_PUBLICATION

options

Combine the options that control the opening of the topic for either publication or subscription. Use MQC.MQSO_* constants to access a topic for subscription and MQC.MQOO_* constants to access a topic for publication.

If more than one option is required, add the values together, or combine the option values using the bitwise OR operator.

alternateUserId

Specify the alternate user ID that is used to check for the required authorization to finish the operation. You must specify *alternateUserId*, if either MQC.MQOO_ALTERNATE_USER_AUTHORITY or MQC.MQSO_ALTERNATE_USER_AUTHORITY is set in the options parameter.

subscriptionName

subscriptionName is required if the options MQC.MQSO_DURABLE or MQC.MQSO_ALTER are provided. In both cases, MQTopic is implicitly opened for subscription. An exception is thrown if the MQC.MQSO_DURABLE is set, and the subscription exists, or if MQC.MQSO_ALTER is set, and the subscription does not exist.

properties

Set any of the special subscription properties listed using a hash table. Specified entries in the hash table are updated with output values. Entries are not added to the hash table to report output values.

- MQC.MQSUB_PROP_ALTERNATE_SECURITY_ID
- MQC.MQSUB_PROP_SUBSCRIPTION_EXPIRY
- MQC.MQSUB_PROP_SUBSCRIPTION_USER_DATA
- MQC.MQSUB_PROP_SUBSCRIPTION_CORRELATION_ID
- MQC.MQSUB_PROP_PUBLICATION_PRIORITY
- MQC.MQSUB_PROP_PUBLICATION_ACCOUNTING_TOKEN
- MQC.MQSUB_PROP_PUBLICATION_APPLICATIONID_DATA

IMQObjectTrigger.NET interface

Implement IMQObjectTrigger to process messages passed by the **runmqdmn**.NET monitor.

Interface

```
public interface IBM.WMQMonitor.IMQObjectTrigger();
```

Depending on whether sync point control is specified in the **runmqdmn** command the message is removed from the queue before or after the Execute method returns.

Methods

void Execute (MQQueueManager *queueManager*, MQQueue *queue*, MQMessage *message*, string *param*);

queueManager

Queue manager hosting the queue being monitored.

queue

Queue being monitored.

message

Message read from the queue.

param

Data passed from UserParameter.

MQC.NET interface

Refer to an MQI constant by prefixing the constant name with MQC.. MQC defines all the constants used by the MQI.

Interface

```
System.Object
└─ IBM.WMQ.MQC
```

```
public interface IBM.WMQ.MQC extends System.Object;
```

Example

```
MQQueue queue;  
queue.closeOptions = MQC.MQCO_DELETE;
```

Character set identifiers for .NET applications

Descriptions of the character sets you can select to encode .NET IBM MQ messages

Character set	Description
37	ibm037
437	ibm437 / PC Original
500	ibm500
819	iso-8859-1 / latin1 / ibm819
1200	Unicode
1208	UTF-8
273	ibm273
277	ibm277
278	ibm278
280	ibm280
284	ibm284
285	ibm285
297	ibm297
420	ibm420
424	ibm424
737	ibm737 / PC Greek
775	ibm775 / PC Baltic
813	iso-8859-7 / greek / ibm813
838	ibm838
850	ibm850 / PC Latin 1

Character set	Description
852	ibm852 / PC Latin 2
855	ibm855 / PC Cyrillic
856	ibm856
857	ibm857 / PC Turkish
860	ibm860 / PC Portuguese
861	ibm861 / PC Icelandic
862	ibm862 / PC Hebrew
863	ibm863 / PC Canadian French
864	ibm864 / PC Arabic
865	ibm865 / PC Nordic
866	ibm866 / PC Russian
868	ibm868
869	ibm869 / PC Modern Greek
870	ibm870
871	ibm871
874	ibm874
875	ibm875
912	iso-8859-2 / latin2 / ibm912
913	iso-8859-3 / latin3 / ibm913
914	iso-8859-4 / latin4 / ibm914
915	iso-8859-5 / cyrillic / ibm915
916	iso-8859-8 / hebrew / ibm916
918	ibm918
920	iso-8859-9 / latin5 / ibm920
921	ibm921
922	ibm922
930	ibm930
932	PC Japanese
933	ibm933
935	ibm935
937	ibm937
939	ibm939
942	ibm942
943	ibm943
948	ibm948

Character set	Description
949	ibm949
950	ibm950 / Big 5 Traditional Chinese
954	EUCJIS
964	ibm964 / CNS 11643 Traditional Chinese
970	ibm970
1006	ibm1006
1025	ibm1025
1026	ibm1026
1089	iso-8859-6 / arabic / ibm1089
1097	ibm1097
1098	ibm1098
1112	ibm1112
1122	ibm1122
1123	ibm1123
1124	ibm1124
1250	Windows Latin 2
1251	Windows Cyrillic
1252	Windows Latin 1
1253	Windows Greek
1254	Windows Turkish
1255	Windows Hebrew
1256	Windows Arabic
1257	Windows Baltic
1258	Windows Vietnamese
1381	ibm1381
1383	ibm1383
2022	JIS
5601	ksc-5601 Korean
33722	ibm33722

IBM MQ C++ classes

The IBM MQ C++ classes encapsulate the IBM MQ Message Queue Interface (MQI). There is a single C++ header file, **imqi.hpp**, which covers all of these classes.

For each class, the following information is shown:

Class hierarchy diagram

A class diagram showing the class in its inheritance relation to its immediate parent classes, if any.

Other relevant classes

Document links to other relevant classes, such as parent classes, and the classes of objects used in method signatures.

Object attributes

Attributes of the class. These are in addition to those attributes defined for any parent classes. Many attributes reflect IBM MQ data-structure members (see [“C++ and MQI cross-reference”](#) on page 1822). For detailed descriptions, see [“Attributes of objects”](#) on page 768.

Constructors

Signatures of the special methods used to create an object of the class.

Object methods (public)

Signatures of methods that require an instance of the class for their operation, and that have no usage restrictions.

Where it applies, the following information is also shown:

Class methods (public)

Signatures of methods that do not require an instance of the class for their operation, and that have no usage restrictions.

Overloaded (parent class) methods

Signatures of those virtual methods that are defined in parent classes, but exhibit different, polymorphic, behavior for this class.

Object methods (protected)

Signatures of methods that require an instance of the class for their operation, and are reserved for use by the implementations of derived classes. This section is of interest only to class writers, as opposed to class users.

Object data (protected)

Implementation details for object instance data available to the implementations of derived classes. This section is of interest only to class writers, as opposed to class users.

Reason codes

MQRC_* values (see [API completion and reason codes](#)) that can be expected from those methods that fail. For an exhaustive list of reason codes that can occur for an object of a class, consult the parent class documentation. The documented list of reason codes for a class does not include the reason codes for parent classes.

Note:

1. Objects of these classes are not thread-safe. This ensures optimal performance, but take care not to access any object from more than one thread.
2. It is recommended that, for a multithreaded program, a separate ImqQueueManager object is used for each thread. Each manager object must have its own independent collection of other objects, ensuring that objects in different threads are isolated from one another.

The classes are:

- [“ImqAuthenticationRecord C++ class”](#) on page 1837
- [“ImqBinary C++ class”](#) on page 1839
- [“ImqCache C++ class”](#) on page 1841
- [“ImqChannel C++ class”](#) on page 1844
- [“ImqCICSBridgeHeader C++ class”](#) on page 1849
- [“ImqDeadLetterHeader C++ class”](#) on page 1855
- [“ImqDistributionList C++ class”](#) on page 1858
- [“ImqError C++ class”](#) on page 1859
- [“ImqGetMessageOptions C++ class”](#) on page 1860
- [“ImqHeader C++ class”](#) on page 1864
- [“ImqIMSBridgeHeader C++ class”](#) on page 1865

- [“ImqItem C++ class” on page 1868](#)
- [“ImqMessage C++ class” on page 1870](#)
- [“ImqMessageTracker C++ class” on page 1876](#)
- [“ImqNamelist C++ class” on page 1879](#)
- [“ImqObject C++ class” on page 1880](#)
- [“ImqProcess C++ class” on page 1886](#)
- [“ImqPutMessageOptions C++ class” on page 1887](#)
- [“ImqQueue C++ class” on page 1890](#)
- [“ImqQueueManager C++ class” on page 1900](#)
- [“ImqReferenceHeader C++ class” on page 1915](#)
- [“ImqString C++ class” on page 1918](#)
- [“ImqTrigger C++ class” on page 1923](#)
- [“ImqWorkHeader C++ class” on page 1926](#)

C++ and MQI cross-reference

This collection of topics contains information relating C++ to the MQI.

Read this information together with [“Data types used in the MQI” on page 231](#).

This table relates MQI data structures to the C++ classes and include files. The following topics show cross-reference information for each C++ class. These cross-references relate to the use of the underlying IBM MQ procedural interfaces. The classes ImqBinary, ImqDistributionList, and ImqString have no attributes that fall into this category and are excluded.

<i>Table 672. Data structure, class, and include-file cross reference</i>		
Data structure	Class	Include file
MQAIR	ImqAuthenticationRecord	imqair.hpp
	ImqBinary	imqbin.hpp
	ImqCache	imqcac.hpp
MQCD	ImqChannel	imqchl.hpp
MQCIH	ImqCICSBridgeHeader	imqcih.hpp
MQDLH	ImqDeadLetterHeader	imqdlh.hpp
MQOR	ImqDistributionList	imqdst.hpp
	ImqError	imqerr.hpp
MQGMO	ImqGetMessageOptions	imqgmo.hpp
	ImqHeader	imqhdr.hpp
MQIIH	ImqIMSBridgeHeader	imqiih.hpp
	ImqItem	imqitm.hpp
MQMD	ImqMessage	imqmsg.hpp
	ImqMessageTracker	imqmtr.hpp
	ImqNamelist	imqnml.hpp
MQOD, MQRR	ImqObject	imqobj.hpp
MQPMO, MQPMR, MQRR	ImqPutMessageOptions	imqpmo.hpp

Table 672. Data structure, class, and include-file cross reference (continued)

Data structure	Class	Include file
	ImqProcess	imqpro.hpp
	ImqQueue	imqqueue.hpp
MQBO, MQCNO, MQCSP	ImqQueueManager	imqmgr.hpp
MQRMH	ImqReferenceHeader	imqrfh.hpp
	ImqString	imqstr.hpp
MQTM	ImqTrigger	imqtrg.hpp
MQTMC		
MQTMC2	ImqTrigger	imqtrg.hpp
MQXQH		
MQWIH	ImqWorkHeader	imqwih.hpp

ImqAuthenticationRecord cross-reference

Cross-reference of attributes, data structures, fields, and calls for the ImqAuthenticationRecord C++ class.

Attribute	Data structure	Field	Call
connection name	MQAIR	AuthInfoConnName	MQCONN
password	MQAIR	LDAPPassword	MQCONN
type	MQAIR	AuthInfoType	MQCONN
user name	MQAIR	LDAPUserNamePtr	MQCONN
	MQAIR	LDAPUserNameOffset	MQCONN
	MQAIR	LDAPUserNameLength	MQCONN

ImqCache cross-reference

Cross-reference of attributes and calls for the ImqCache C++ class.

Attribute	Call
automatic buffer	MQGET
buffer length	MQGET
buffer pointer	MQGET, MQPUT
data length	MQGET
data offset	MQGET
data pointer	MQGET
message length	MQGET, MQPUT

ImqChannel cross-reference

Cross-reference of attributes, data structures, fields, and calls for the ImqChannel C++ class.

Attribute	Data structure	Field	Call
batch heart-beat	MQCD	BatchHeartbeat	MQCONN
channel name	MQCD	ChannelName	MQCONN
connection name	MQCD	ConnectionName	MQCONN
	MQCD	ShortConnectionName	MQCONN
header compression	MQCD	HdrCompList	MQCONN
heart-beat interval	MQCD	HeartbeatInterval	MQCONN
keep alive interval	MQCD	KeepAliveInterval	MQCONN
local address	MQCD	LocalAddress	MQCONN
maximum message length	MQCD	MaxMsgLength	MQCONN
message compression	MQCD	MsgCompList	MQCONN
mode name	MQCD	ModeName	MQCONN
password	MQCD	Password	MQCONN
receive exit count	MQCD		MQCONN
receive exit names	MQCD	ReceiveExit	MQCONN
	MQCD	ReceiveExitsDefined	MQCONN
	MQCD	ReceiveExitPtr	MQCONN
receive user data	MQCD	ReceiveUserData	MQCONN
	MQCD	ReceiveUserDataPtr	MQCONN
security exit name	MQCD	SecurityExit	MQCONN
security user data	MQCD	SecurityUserData	MQCONN
send exit count	MQCD		MQCONN
send exit names	MQCD	SendExit	MQCONN
	MQCD	SendExitsDefined	MQCONN
	MQCD	SendExitPtr	MQCONN
send user data	MQCD	SendUserData	MQCONN
	MQCD	SendUserDataPtr	MQCONN
SSL CipherSpec	MQCD	sslCipherSpecification	MQCONN
SSL client authentication type	MQCD	sslClientAuthentication	MQCONN
SSL peer name	MQCD	sslPeerName	MQCONN
transaction program name	MQCD	TpName	MQCONN
transport type	MQCD	TransportType	MQCONN
user id	MQCD	UserIdentifier	MQCONN

ImqCICSBridgeHeader cross-reference

Cross-reference of attributes, data structures, and fields for the ImqCICSBridgeHeader C++ class.

Attribute	Data structure	Field
bridge abend code	MQCIH	AbendCode
ADS descriptor	MQCIH	AdsDescriptor
attention identifier	MQCIH	AttentionId
authenticator	MQCIH	Authenticator
bridge completion code	MQCIH	BridgeCompletionCode
bridge error offset	MQCIH	ErrorOffset
bridge reason code	MQCIH	BridgeReason
bridge cancel code	MQCIH	CancelCode
conversational task	MQCIH	ConversationalTask
cursor position	MQCIH	CursorPosition
facility token	MQCIH	Facility
facility keep time	MQCIH	FacilityKeepTime
facility like	MQCIH	FacilityLike
function	MQCIH	Function
get wait interval	MQCIH	GetWaitInterval
link type	MQCIH	LinkType
next transaction identifier	MQCIH	NextTransactionId
output data length	MQCIH	OutputDataLength
reply-to format	MQCIH	ReplyToFormat
bridge return code	MQCIH	ReturnCode
start code	MQCIH	StartCode
task end status	MQCIH	TaskEndStatus
transaction identifier	MQCIH	TransactionId
uow control	MQCIH	UowControl
version	MQCIH	Version

ImqDeadLetterHeader cross reference

Cross-reference of attributes, data structures, and fields for the ImqDeadLetterHeader C++ class.

Attribute	Data structure	Field
dead-letter reason code	MQDLH	Reason
destination queue manager name	MQDLH	DestQMgrName
destination queue name	MQDLH	DestQName
put application name	MQDLH	PutApplName
put application type	MQDLH	PutApplType

Attribute	Data structure	Field
put date	MQDLH	PutDate
put time	MQDLH	PutTime

ImqError cross reference

Cross-reference of attributes and calls for the ImqError C++ class.

Attribute	Call
completion code	MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET
reason code	MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET

ImqGetMessageOptions cross reference

Cross-reference of attributes, data structures, and fields for the ImqGetMessageOptions C++ class.

Attribute	Data structure	Field
group status	MQGMO	GroupStatus
match options	MQGMO	MatchOptions
message token	MQGMO	MessageToken
options	MQGMO	Options
resolved queue name	MQGMO	ResolvedQName
returned length	MQGMO	ReturnedLength
segmentation	MQGMO	Segmentation
segment status	MQGMO	SegmentStatus
	MQGMO	Signal1
	MQGMO	Signal2
syncpoint participation	MQGMO	Options
wait interval	MQGMO	WaitInterval

ImqHeader cross reference

Cross-reference of attributes, data structures, and fields for the ImqHeader C++ class.

Attribute	Data structure	Field
character set	MQDLH, MQIIH	CodedCharSetId
encoding	MQDLH, MQIIH	Encoding
format	MQDLH, MQIIH	Format
header flags	MQIIH, MQRMH	Flags

ImqIMSBridgeHeader cross reference

Cross-reference of attributes, data structures, and fields for the ImqAuthenticationRecord C++ class.

Attribute	Data structure	Field
authenticator	MQIIH	Authenticator
commit mode	MQIIH	CommitMode
logical terminal override	MQIIH	LTermOverride
message format services map name	MQIIH	MFSMapName
reply-to format	MQIIH	ReplyToFormat
security scope	MQIIH	SecurityScope
transaction instance id	MQIIH	TranInstanceId
transaction state	MQIIH	TranState

ImqItem cross reference

Cross-reference of attributes and calls for the ImqItem C++ class.

Attribute	Call
structure id	MQGET

ImqMessage cross reference

Cross-reference of attributes, data structures, fields, and calls for the ImqMessage C++ class.

Attribute	Data structure	Field	Call
application id data	MQMD	ApplIdentityData	
application origin data	MQMD	ApplOriginData	
backout count	MQMD	BackoutCount	
character set	MQMD	CodedCharSetId	
encoding	MQMD	Encoding	
expiry	MQMD	Expiry	
format	MQMD	Format	
message flags	MQMD	MsgFlags	
message type	MQMD	MsgType	
offset	MQMD	Offset	
original length	MQMD	OriginalLength	
persistence	MQMD	Persistence	
priority	MQMD	Priority	
put application name	MQMD	PutApplName	
put application type	MQMD	PutApplType	
put date	MQMD	PutDate	

Attribute	Data structure	Field	Call
put time	MQMD	PutTime	
reply-to queue manager name	MQMD	ReplyToQMgr	
reply-to queue name	MQMD	ReplyToQ	
report	MQMD	Report	
sequence number	MQMD	MsgSeqNumber	
total message length		DataLength	MQGET
user id	MQMD	UserIdentifier	

ImqMessageTracker cross reference

Cross-reference of attributes, data structures, and fields for the ImqMessageTracker C++ class.

Attribute	Data structure	Field
accounting token	MQMD	AccountingToken
correlation id	MQMD	CorrelId
feedback	MQMD	Feedback
group id	MQMD	GroupId
message id	MQMD	MsgId

ImqNamelist cross reference

Cross-reference of attributes, inquiries, and calls for the ImqNamelist C++ class.

Attribute	Inquiry	Call
name count	MQIA_NAME_COUNT	MQINQ
namelist name	MQCA_NAMELIST_NAME	MQINQ

ImqObject cross reference

Cross-reference of attributes, data structures, fields, inquiries, and calls for the ImqObject C++ class.

Attribute	Data structure	Field	Inquiry	Call
alteration date			MQCA_ALTERATION_DATE	MQINQ
alteration time			MQCA_ALTERATION_TIME	MQINQ
alternate user id	MQOD	AlternateUserId		
alternate security id				
close options				MQCLOSE
description			MQCA_Q_DESC, MQCA_Q_MGR_DESC, MQCA_PROCESS_DESC	MQINQ
name	MQOD	ObjectName	MQCA_Q_MGR_NAME, MQCQ_Q_NAME, MQCA_PROCESS_NAME	MQINQ

Attribute	Data structure	Field	Inquiry	Call
open options				MQOPEN
open status				MQOPEN, MQCLOSE
queue manager identifier	queue manager identifier		MQCA_Q_MGR_IDENTIFIER	MQINQ

ImqProcess cross-reference

Cross-reference of attributes, inquiries, and calls for the ImqAuthenticationRecord C++ class.

Attribute	Inquiry	Call
application id	MQCA_APPL_ID	MQINQ
application type	MQIA_APPL_TYPE	MQINQ
environment data	MQCA_ENV_DATA	MQINQ
user data	MQCA_USER_DATA	MQINQ

ImqPutMessageOptions cross-reference

Cross-reference of attributes, data structures, and fields for the ImqAuthenticationRecord C++ class.

Table 673. ImqPutMessageOptions cross reference		
Attribute	Data structure	Field
context reference	MQPMO	Context
	MQPMO	InvalidDestCount
	MQPMO	KnownDestCount
options	MQPMO	Options
record fields	MQPMO	PutMsgRecFields
resolved queue manager name	MQPMO	ResolvedQMgrName
resolved queue name	MQPMO	ResolvedQName
	MQPMO	Timeout
	MQPMO	UnknownDestCount
syncpoint participation	MQPMO	Options

ImqQueue cross-reference

Cross-reference of attributes, data structures, fields, inquiries, and calls for the ImqQueue C++ class.

Table 674. ImqQueue cross reference				
Attribute	Data structure	Field	Inquiry	Call
backout requeue name			MQCA_BACKOUT_REQ_Q_NAME	MQINQ
backout threshold			MQIA_BACKOUT_THRESHOLD	MQINQ

Table 674. ImqQueue cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
base queue name			MQCA_BASE_Q_NAME	MQINQ
cluster name			MQCA_CLUSTER_NAME	MQINQ
cluster namelist name			MQCA_CLUSTER_NAMELIST	MQINQ
cluster workload rank			MQIA_CLWL_Q_RANK	MQINQ
cluster workload priority			MQIA_CLWL_Q_PRIORITY	MQINQ
cluster workload use queue			MQIA_CLWL_USEQ	MQINQ
creation date			MQCA_CREATION_DATE	MQINQ
creation time			MQCA_CREATION_TIME	MQINQ
current depth			MQIA_CURRENT_Q_DEPTH	MQINQ
default bind			MQIA_DEF_BIND	MQINQ
default input open option			MQIA_DEF_INPUT_OPEN_OPTION	MQINQ
default persistence			MQIA_DEF_PERSISTENCE	MQINQ
default priority			MQIA_DEF_PRIORITY	MQINQ
definition type			MQIA_DEFINITION_TYPE	MQINQ
depth high event			MQIA_Q_DEPTH_HIGH_EVENT	MQINQ
depth high limit			MQIA_Q_DEPTH_HIGH_LIMIT	MQINQ
depth low event			MQIA_Q_DEPTH_LOW_EVENT	MQINQ
depth low limit			MQIA_Q_DEPTH_LOW_LIMIT	MQINQ
depth maximum event			MQIA_Q_DEPTH_MAX_LIMIT	MQINQ
distribution lists			MQIA_DIST_LISTS	MQINQ, MQSET
dynamic queue name	MQOD	DynamicQName		
harden get backout			MQIA_HARDEN_GET_BACKOUT	MQINQ
index type			MQIA_INDEX_TYPE	MQINQ
inhibit get			MQIA_INHIBIT_GET	MQINQ, MQSET
inhibit put			MQIA_INHIBIT_PUT	MQINQ, MQSET
initiation queue name			MQCA_INITIATION_Q_NAME	MQINQ
maximum depth			MQIA_MAX_Q_DEPTH	MQINQ
maximum message length			MQIA_MAX_MSG_LENGTH	MQINQ

Table 674. *ImqQueue* cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
message delivery sequence			MQIA_MSG_DELIVERY_SEQUENCE	MQINQ
next distributed queue				
non persistent message class			MQIA_NPM_CLASS	MQINQ
open input count			MQIA_OPEN_INPUT_COUNT	MQINQ
open output count			MQIA_OPEN_OUTPUT_COUNT	MQINQ
previous distributed queue				
process name			MQCA_PROCESS_NAME	MQINQ
queue accounting			MQIA_ACCOUNTING_Q	MQINQ
queue manager name	MQOD	ObjectQMgrName		
queue monitoring			MQIA_MONITORING_Q	MQINQ
queue statistics			MQIA_STATISTICS_Q	MQINQ
queue type			MQIA_Q_TYPE	MQINQ
remote queue manager name			MQCA_REMOTE_Q_MGR_NAME	MQINQ
remote queue name			MQCA_REMOTE_Q_NAME	MQINQ
resolved queue manager name	MQOD	ResolvedQMgrName		
resolved queue name	MQOD	ResolvedQName		
retention interval			MQIA_RETENTION_INTERVAL	MQINQ
scope			MQIA_SCOPE	MQINQ
service interval			MQIA_Q_SERVICE_INTERVAL	MQINQ
service interval event			MQIA_Q_SERVICE_INTERVAL_EVENT	MQINQ
shareability			MQIA_SHAREABILITY	MQINQ
storage class			MQCA_STORAGE_CLASS	MQINQ
transmission queue name			MQCA_XMIT_Q_NAME	MQINQ
trigger control			MQIA_TRIGGER_CONTROL	MQINQ, MQSET
trigger data			MQCA_TRIGGER_DATA	MQINQ, MQSET
trigger depth			MQIA_TRIGGER_DEPTH	MQINQ, MQSET
trigger message priority			MQIA_TRIGGER_MSG_PRIORITY	MQINQ, MQSET

Table 674. ImqQueue cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
trigger type			MQIA_TRIGGER_TYPE	MQINQ, MQSET
usage			MQIA_USAGE	MQINQ

ImqQueueManager cross-reference

Cross-reference of attributes, data structures, fields, inquiries, and calls for the ImqQueueManager C++ class.

Attribute	Data structure	Field	Inquiry	Call
accounting connections override			MQIA_ACCOUNTING_CONN_OVERRIDE	MQINQ
accounting interval			MQIA_ACCOUNTING_INTERVAL	MQINQ
activity recording			MQIA_ACTIVITY_RECORDING	MQINQ
adopt new mca check			MQIA_ADOPTNEWMCA_CHECK	MQINQ
adopt new mca type			MQIA_ADOPTNEWMCA_TYPE	MQINQ
authentication type	MQCSP	AuthenticationType		MQCONN
authority event			MQIA_AUTHORITY_EVENT	MQINQ
begin options	MQBO	Options		MQBEGIN
bridge event			MQIA_BRIDGE_EVENT	MQINQ
channel auto definition			MQIA_CHANNEL_AUTO_DEF	MQINQ
channel auto definition event			MQIA_CHANNEL_AUTO_EVENT	MQIA
channel auto definition exit			MQIA_CHANNEL_AUTO_EXIT	MQIA
channel event			MQIA_CHANNEL_EVENT	MQINQ
channel initiator adapters			MQIA_CHINIT_ADAPTERS	MQINQ
channel initiator control			MQIA_CHINIT_CONTROL	MQINQ
channel initiator dispatchers			MQIA_CHINIT_DISPATCHERS	MQINQ
channel initiator trace auto start			MQIA_CHINIT_TRACE_AUTO_START	MQINQ
channel initiator trace table size			MQIA_CHINIT_TRACE_TABLE_SIZE	MQINQ

Attribute	Data structure	Field	Inquiry	Call
channel monitoring			MQIA_MONITORING_CHANNEL	MQINQ
channel reference	MQCD	ChannelType		MQCONN
channel statistics			MQIA_STATISTICS_CHANNEL	MQINQ
character set			MQIA_CODED_CHAR_SET_ID	MQINQ
cluster sender monitoring			MQIA_MONITORING_AUTO_CLUSSDR	MQINQ
cluster sender statistics			MQIA_STATISTICS_AUTO_CLUSSDR	MQINQ
cluster workload data			MQCA_CLUSTER_WORKLOAD_DATA	MQINQ
cluster workload exit			MQCA_CLUSTER_WORKLOAD_EXIT	MQINQ
cluster workload length			MQIA_CLUSTER_WORKLOAD_LENGTH	MQINQ
cluster workload mru			MQIA_CLWL_MRU_CHANNELS	MQINQ
cluster workload use queue			MQIA_CLWL_USEQ	MQINQ
command event			MQIA_COMMAND_EVENT	MQINQ
command input queue name			MQCA_COMMAND_INPUT_Q_NAME	MQINQ
command level			MQIA_COMMAND_LEVEL	MQINQ
command server control			MQIA_CMD_SERVER_CONTROL	MQINQ
connect options	MQCNO	Options		MQCONN, MQCONN
connection id	MQCNO	ConnectionId		MQCONN
connection status				MQCONN, MQCONN, MQDISC
connection tag	MQCD	ConnTag		MQCONN
cryptographic hardware	MQSCO	CryptoHardware		MQCONN
dead-letter queue name			MQCA_DEAD_LETTER_Q_NAME	MQINQ
default transmission queue name			MQCA_DEF_XMIT_Q_NAME	MQINQ
distribution lists			MQIA_DIST_LISTS	MQINQ
dns group			MQCA_DNS_GROUP	MQINQ

Attribute	Data structure	Field	Inquiry	Call
dns wlm			MQIA_DNS_WLM	MQINQ
first authentication record	MQSCO	AuthInfoRecOffset		MQCONN
	MQSCO	AuthInfoRecPtr		MQCONN
inhibit event			MQIA_INHIBIT_EVENT	MQINQ
ip address version			MQIA_IP_ADDRESS_VERSION	MQINQ
key repository	MQSCO	KeyRepository		MQCONN
key reset count	MQSCO	KeyResetCount		MQCONN
listener timer			MQIA_LISTENER_TIMER	MQINQ
local event			MQIA_LOCAL_EVENT	MQINQ
logger event			MQIA_LOGGER_EVENT	MQINQ
lu group name			MQCA_LU_GROUP_NAME	MQINQ
lu name			MQCA_LU_NAME	MQINQ
lu62 arm suffix			MQCA_LU62_ARM_SUFFIX	MQINQ
lu62 channels			MQIA_LU62_CHANNELS	MQINQ
maximum active channels			MQIA_ACTIVE_CHANNELS	MQINQ
maximum channels			MQIA_MAX_CHANNELS	MQINQ
maximum handles			MQIA_MAX_HANDLES	MQINQ
maximum message length			MQIA_MAX_MSG_LENGTH	MQINQ
maximum priority			MQIA_MAX_PRIORITY	MQINQ
maximum uncommitted messages			MQIA_MAX_UNCOMMITTED_MSGS	MQINQ
mqi accounting			MQIA_ACCOUNTING_MQI	MQINQ
mqi statistics			MQIA_STATISTICS_MQI	MQINQ
outbound port maximum			MQIA_OUTBOUND_PORT_MAX	MQINQ
outbound port minimum			MQIA_OUTBOUND_PORT_MIN	MQINQ
password	MQCSP	CSPPasswordPtr		MQCONN
	MQCSP	CSPPasswordOffset		MQCONN
	MQCSP	CSPPasswordLength		MQCONN
performance event			MQIA_PERFORMANCE_EVENT	MQINQ

Attribute	Data structure	Field	Inquiry	Call
platform			MQIA_PLATFORM	MQINQ
queue accounting			MQIA_ACCOUNTING_Q	MQINQ
queue monitoring			MQIA_MONITORING_Q	MQINQ
queue statistics			MQIA_STATISTICS_Q	MQINQ
receive timeout			MQIA_RECEIVE_TIMEOUT	MQINQ
receive timeout minimum			MQIA_RECEIVE_TIMEOUT_MIN	MQINQ
receive timeout type			MQIA_RECEIVE_TIMEOUT_TYPE	MQINQ
remote event			MQIA_REMOTE_EVENT	MQINQ
repository name			MQCA_REPOSITORY_NAME	MQINQ
repository namelist			MQCA_REPOSITORY_NAMELIST	MQINQ
shared queue queue manager name			MQIA_SHARED_Q_Q_MGR_NAME	MQINQ
ssl event			MQIA_SSL_EVENT	MQINQ
ssl fips			MQIA_SSL_FIPS_REQUIRED	MQINQ
ssl key reset count			MQIA_SSL_RESET_COUNT	MQINQ
start-stop event			MQIA_START_STOP_EVENT	MQINQ
statistics interval			MQIA_STATISTICS_INTERVAL	MQINQ
syncpoint availability			MQIA_SYNCPOINT	MQINQ
tcp channels			MQIA_TCP_CHANNELS	MQINQ
tcp keep alive			MQIA_TCP_KEEP_ALIVE	MQINQ
tcp name			MQCA_TCP_NAME	MQINQ
tcp stack type			MQIA_TCP_STACK_TYPE	MQINQ
trace route recording			MQIA_TRACE_ROUTE_RECORDING	MQINQ
trigger interval			MQIA_TRIGGER_INTERVAL	MQINQ
user id	MQCSP	CSPUserIdPtr		MQCONN
	MQCSP	CSPUserIdOffset		MQCONN
	MQCSP	CSPUserIdLength		MQCONN

ImqReferenceHeader cross-reference

Cross-reference of attributes, data structures, and fields for the ImqAuthenticationRecord C++ class.

Attribute	Data structure	Field
destination environment	MQRMH	DestEnvLength, DestEnvOffset
destination name	MQRMH	DestNameLength, DestNameOffset
instance id	MQRMH	ObjectInstanceId
logical length	MQRMH	DataLogicalLength
logical offset	MQRMH	DataLogicalOffset
logical offset 2	MQRMH	DataLogicalOffset2
reference type	MQRMH	ObjectType
source environment	MQRMH	SrcEnvLength, SrcEnvOffset
source name	MQRMH	SrcNameLength, SrcNameOffset

ImqTrigger cross-reference

Cross-reference of attributes, data structures, and fields for the ImqAuthenticationRecord C++ class.

<i>Table 675. ImqTrigger cross reference</i>		
Attribute	Data structure	Field
application id	MQTM	ApplId
application type	MQTM	ApplType
environment data	MQTM	EnvData
process name	MQTM	ProcessName
queue name	MQTM	QName
trigger data	MQTM	TriggerData
user data	MQTM	UserData

ImqWorkHeader cross-reference

Cross-reference of attributes, data structures, and fields for the ImqAuthenticationRecord C++ class.

Attribute	Data structure	Field
message token	MQWIIH	MessageToken
service name	MQWIIH	ServiceName
service step	MQWIIH	ServiceStep

ImqAuthenticationRecord C++ class

This class encapsulates an authentication information record (MQAIR) for use during execution of the `ImqQueueManager::connect` method, for custom SSL client connections.

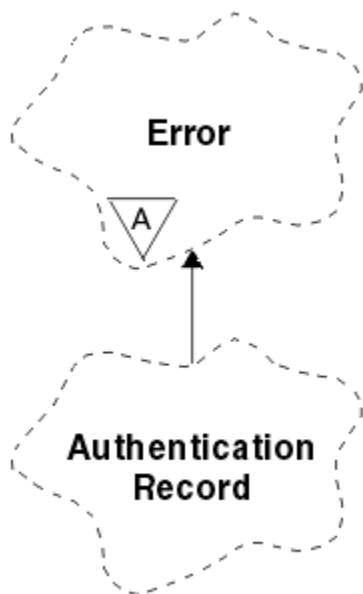


Figure 46. `ImqAuthenticationRecord` class

See the description of the `ImqQueueManager::connect` method for more details. This class is not available on the z/OS platform.

- [“Object attributes” on page 1837](#)
- [“Constructors” on page 1838](#)
- [“Object methods \(public\)” on page 1838](#)
- [“Object methods \(protected\)” on page 1838](#)

Object attributes

connection name

The name of the connection to the LDAP CRL server. This is the IP address or DNS name, followed optionally by the port number, in parentheses.

connection reference

A reference to an `ImqQueueManager` object that provides the required connection to a (local) queue manager. The initial value is zero. Do not confuse this with the queue manager name that identifies a queue manager (possibly remote) for a named queue.

next authentication record

Next object of this class, in no particular order, having the same **connection reference** as this object. The initial value is zero.

password

A password supplied for connection authentication to the LDAP CRL server.

previous authentication record

Previous object of this class, in no particular order, having the same **connection reference** as this object. The initial value is zero.

type

The type of authentication information contained in the record.

user name

A user identifier supplied for authorization to the LDAP CRL server.

Constructors

ImqAuthenticationRecord();

The default constructor.

Object methods (public)

void operator = (const ImqAuthenticationRecord & *air*);

Copies instance data from *air*, replacing the existing instance data.

const ImqString & connectionName () const ;

Returns the **connection name**.

void setConnectionName (const ImqString & *name*);

Sets the **connection name**.

void setConnectionName (const char * *name* = 0);

Sets the **connection name**.

ImqQueueManager * connectionReference () const ;

Returns the **connection reference**.

void setConnectionReference (ImqQueueManager & *manager*);

Sets the **connection reference**.

void setConnectionReference (ImqQueueManager * *manager* = 0);

Sets the **connection reference**.

void copyOut (MQAIR * *pAir*);

Copies instance data to *pAir*, replacing the existing instance data. This might involve allocating dependent storage.

void clear (MQAIR * *pAir*);

Clears the structure and releases dependent storage referenced by *pAir*.

ImqAuthenticationRecord * nextAuthenticationRecord () const ;

Returns the **next authentication record**.

const ImqString & password () const ;

Returns the **password**.

void setPassword (const ImqString & *password*);

Sets the **password**.

void setPassword (const char * *password* = 0);

Sets the **password**.

ImqAuthenticationRecord * previousAuthenticationRecord () const ;

Returns the **previous authentication record**.

MQLONG type () const ;

Returns the **type**.

void setType (const MQLONG *type*);

Sets the **type**.

const ImqString & userName () const ;

Returns the **user name**.

void setUsername (const ImqString & *name*);

Sets the **user name**.

void setUsername (const char * *name* = 0);

Sets the **user name**.

Object methods (protected)

void setNextAuthenticationRecord (ImqAuthenticationRecord * *pAir* = 0);

Sets the **next authentication record**.

Attention: Use this function only if you are sure that it will not break the authentication record list.

void setPreviousAuthenticationRecord (ImqAuthenticationRecord * pAir = 0);

Sets the **previous authentication record**.

Attention: Use this function only if you are sure that it will not break the authentication record list.

ImqBinary C++ class

This class encapsulates a binary byte array that can be used for ImqMessage **accounting token**, **correlation id**, and **message id** values. It allows easy assignment, copying, and comparison.

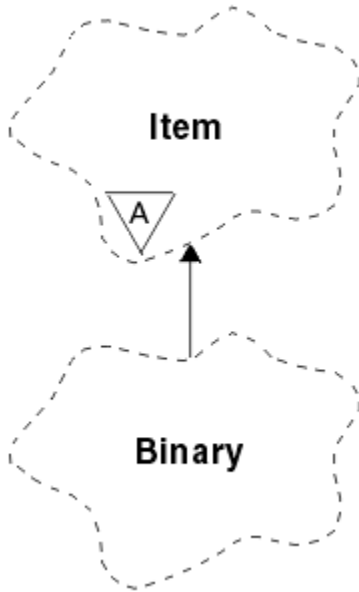


Figure 47. ImqBinary class

- [“Object attributes” on page 1839](#)
- [“Constructors” on page 1839](#)
- [“Overloaded ImqItem methods” on page 1840](#)
- [“Object methods \(public\)” on page 1840](#)
- [“Object methods \(protected\)” on page 1840](#)
- [“Reason codes” on page 1840](#)

Object attributes

data

An array of bytes of binary data. The initial value is null.

data length

The number of bytes. The initial value is zero.

data pointer

The address of the first byte of the **data**. The initial value is zero.

Constructors

ImqBinary();

The default constructor.

ImqBinary(const ImqBinary & binary);

The copy constructor.

ImqBinary(const void * data, const size_t length);

Copies *length* bytes from *data*.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & msg);

Copies the **data** to the message buffer, replacing any existing content. Sets the *msg format* to MQFMT_NONE.

See the ImqItem class method description for further details.

virtual ImqBoolean pasteIn (ImqMessage & msg);

Sets the **data** by transferring the remaining data from the message buffer, replacing the existing **data**.

To be successful, the ImqMessage **format** must be MQFMT_NONE.

See the ImqItem class method description for further details.

Object methods (public)

void operator = (const ImqBinary & binary);

Copies bytes from *binary*.

ImqBoolean operator == (const ImqBinary & binary);

Compares this object with *binary*. It returns FALSE if not equal and TRUE otherwise. The objects are equal if they have the same **data length** and the bytes match.

ImqBoolean copyOut (void * buffer, const size_t length, const char pad = 0);

Copies up to *length* bytes from the **data pointer** to *buffer*. If the **data length** is insufficient, the remaining space in *buffer* is filled with *pad* bytes. *buffer* can be zero if *length* is also zero. *length* must not be negative. It returns TRUE if successful.

size_t dataLength () const ;

Returns the **data length**.

ImqBoolean setDataLength (const size_t length);

Sets the **data length**. If the **data length** is changed as a result of this method, the data in the object is uninitialized. It returns TRUE if successful.

void * dataPointer () const ;

Returns the **data pointer**.

ImqBoolean isNull () const ;

Returns TRUE if the **data length** is zero, or if all the **data** bytes are zero. Otherwise it returns FALSE.

ImqBoolean set (const void * buffer, const size_t length);

Copies *length* bytes from *buffer*. It returns TRUE if successful.

Object methods (protected)

void clear ();

Reduces the **data length** to zero.

Reason codes

- MQRC_NO_BUFFER
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_INCONSISTENT_FORMAT

ImqCache C++ class

Use this class to hold or marshal data in memory.

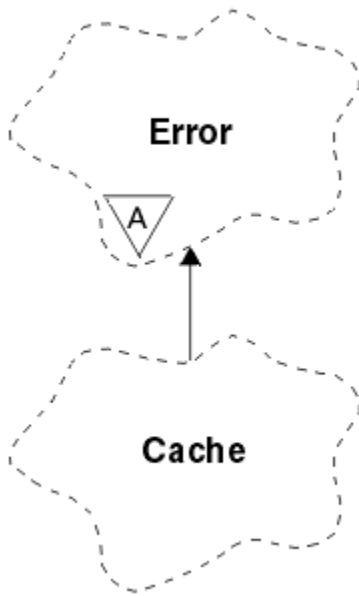


Figure 48. ImqCache class

Use this class to hold or marshal data in memory. You can nominate a buffer of memory of fixed size, or the system can provide a flexible amount of memory automatically. This class relates to the MQI calls listed in [“ImqCache cross-reference” on page 1823](#).

- [“Object attributes” on page 1841](#)
- [“Constructors” on page 1842](#)
- [“Object methods \(public\)” on page 1842](#)
- [“Reason codes” on page 1843](#)

Object attributes

automatic buffer

Indicates whether buffer memory is managed automatically by the system (TRUE) or is supplied by the user (FALSE). It is initially set to TRUE.

This attribute is not set directly. It is set indirectly using either the **useEmptyBuffer** or the **useFullBuffer** method.

If user storage is supplied, this attribute is FALSE, buffer memory cannot grow, and buffer overflow errors can occur. The address and length of the buffer remain constant.

If user storage is not supplied, this attribute is TRUE, and buffer memory can grow incrementally to accommodate an arbitrary amount of message data. However, when the buffer grows, the address of the buffer might change, so be careful when using the **buffer pointer** and **data pointer**.

buffer length

The number of bytes of memory in the buffer. The initial value is zero.

buffer pointer

The address of the buffer memory. The initial value is null.

data length

The number of bytes succeeding the **data pointer**. This must be equal to or less than the **message length**. The initial value is zero.

data offset

The number of bytes preceding the **data pointer**. This must be equal to or less than the **message length**. The initial value is zero.

data pointer

The address of the part of the buffer that is to be written to or read from next. The initial value is null.

message length

The number of bytes of significant data in the buffer. The initial value is zero.

Constructors**ImqCache();**

The default constructor.

ImqCache(const ImqCache & cache);

The copy constructor.

Object methods (public)**void operator = (const ImqCache & cache);**

Copies up to **message length** bytes of data from the *cache* object to the object. If **automatic buffer** is FALSE, the **buffer length** must already be sufficient to accommodate the copied data.

ImqBoolean automaticBuffer () const ;

Returns the **automatic buffer** value.

size_t bufferLength () const ;

Returns the **buffer length**.

char * bufferPointer () const ;

Returns the **buffer pointer**.

void clearMessage ();

Sets the **message length** and **data offset** to zero.

size_t dataLength () const ;

Returns the **data length**.

size_t dataOffset () const ;

Returns the **data offset**.

ImqBoolean setDataOffset (const size_t offset);

Sets the **data offset**. The **message length** is increased if necessary to ensure that it is no less than the **data offset**. This method returns TRUE if successful.

char * dataPointer () const ;

Returns a copy of the **data pointer**.

size_t messageLength () const ;

Returns the **message length**.

ImqBoolean setMessageLength (const size_t length);

Sets the **message length**. Increases the **buffer length** if necessary to ensure that the **message length** is no greater than the **buffer length**. Reduces the **data offset** if necessary to ensure that it is no greater than the **message length**. It returns TRUE if successful.

ImqBoolean moreBytes (const size_t bytes-required);

Assures that *bytes-required* more bytes are available (for writing) between the **data pointer** and the end of the buffer. It returns TRUE if successful.

If **automatic buffer** is TRUE, more memory is acquired as required; otherwise, the **buffer length** must already be adequate.

ImqBoolean read (const size_t length, char * & external-buffer);

Copies *length* bytes, from the buffer starting at the **data pointer** position, into the *external-buffer*.

After the data has been copied, the **data offset** is increased by *length*. This method returns TRUE if successful.

ImqBoolean resizeBuffer (const size_t length);

Varies the **buffer length**, provided that **automatic buffer** is TRUE. This is achieved by reallocating the buffer memory. Up to **message length** bytes of data from the existing buffer are copied to the new one. The maximum number copied is *length* bytes. The **buffer pointer** is changed. The **message length** and **data offset** are preserved as closely as possible within the confines of the new buffer. It returns TRUE if successful, and FALSE if **automatic buffer** is FALSE.

Note: This method can fail with MQRC_STORAGE_NOT_AVAILABLE if there is any problem with system resources.

ImqBoolean useEmptyBuffer (const char * external-buffer, const size_t length);

Identifies an empty user buffer, setting the **buffer pointer** to point to *external-buffer*, the **buffer length** to *length*, and the **message length** to zero. Performs a **clearMessage**. If the buffer is fully primed with data, use the **useFullBuffer** method instead. If the buffer is partially primed with data, use the **setMessageLength** method to indicate the correct amount. This method returns TRUE if successful.

This method can be used to identify a fixed amount of memory, as described previously (*external-buffer* is not null and *length* is nonzero), in which case **automatic buffer** is set to FALSE, or it can be used to revert to system-managed flexible memory (*external-buffer* is null and *length* is zero), in which case **automatic buffer** is set to TRUE.

ImqBoolean useFullBuffer (const char * externalBuffer, const size_t length);

As for **useEmptyBuffer**, except that the **message length** is set to *length*. It returns TRUE if successful.

ImqBoolean write (const size_t length, const char * external-buffer);

Copies *length* bytes, from the *external-buffer*, into the buffer starting at the **data pointer** position. After the data has been copied, the **data offset** is increased by *length*, and the **message length** is increased if necessary to ensure that it is no less than the new **data offset** value. This method returns TRUE if successful.

If **automatic buffer** is TRUE, an adequate amount of memory is guaranteed; otherwise, the ultimate **data offset** must not exceed the **buffer length**.

Reason codes

- MQRC_BUFFER_NOT_AUTOMATIC
- MQRC_DATA_TRUNCATED
- MQRC_INSUFFICIENT_BUFFER
- MQRC_INSUFFICIENT_DATA
- MQRC_NULL_POINTER
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_ZERO_LENGTH

ImqChannel C++ class

This class encapsulates a channel definition (MQCD) for use during execution of the `Manager::connect` method, for custom client connections.

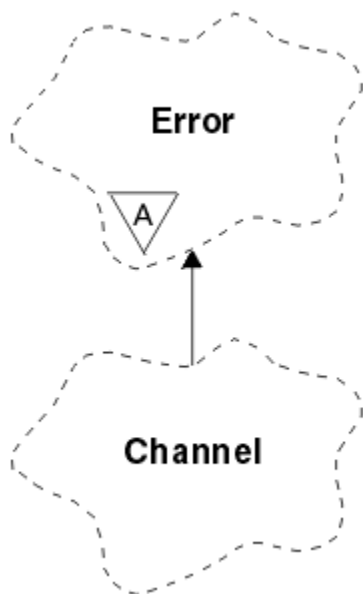


Figure 49. *ImqChannel* class

See the description of the `Manager::connect` method, and [Sample program HELLO WORLD \(imqwrlid.cpp\)](#), for more details.

Not all the listed methods are applicable to all platforms. See the descriptions of the [DEFINE CHANNEL](#) and [ALTER CHANNEL](#) commands for more information.

The `ImqChannel` class is not supported on z/OS.

- [“Object attributes” on page 1844](#)
- [“Constructors” on page 1845](#)
- [“Object methods \(public\)” on page 1846](#)
- [“Reason codes” on page 1849](#)

Object attributes

batch heart-beat

The number of milliseconds between checks that a remote channel is active. The initial value is 0.

channel name

The name of the channel. The initial value is null.

connection name

The name of the connection. For example, the IP address of a host computer. The initial value is null.

header compression

The list of header data compression techniques supported by the channel. The initial values are all set to `MQCOMPRESS_NOT_AVAILABLE`.

heart-beat interval

The number of seconds between checks that a connection is still working. The initial value is 300.

keep alive interval

The number of seconds passed to the communications stack specifying the keep alive timing for the channel. The initial value is `MQKAI_AUTO`.

local address

The local communications address for the channel.

maximum message length

The maximum length of message supported by the channel in a single communication. The initial value is 4 194 304.

message compression

The list of message data compression techniques supported by the channel. The initial values are all set to MQCOMPRESS_NOT_AVAILABLE.

mode name

The name of the mode. The initial value is null.

password

A password supplied for connection authentication. The initial value is null.

receive exit count

The number of receive exits. The initial value is zero. This attribute is read-only.

receive exit names

The names of receive exits.

receive user data

Data associated with receive exits.

security exit name

The name of a security exit to be invoked on the server side of the connection. The initial value is null.

security user data

Data to be passed to the security exit. The initial value is null.

send exit count

The number of send exits. The initial value is zero. This attribute is read-only.

send exit names

The names of send exits.

send user data

Data associated with send exits.

SSL CipherSpec

CipherSpec for use with SSL.

SSL client authentication type

Client authentication type for use with SSL.

SSL peer name

Peer name for use with SSL.

transaction program name

The name of the transaction program. The initial value is null.

transport type

The transport type of the connection. The initial value is MQXPT_LU62.

user id

A user identifier supplied for authorization. The initial value is null.

Constructors**ImqChannel() ;**

The default constructor.

ImqChannel(const ImqChannel & channel);

The copy constructor.

Object methods (public)

void operator = (const ImqChannel & *channel*);

Copies instance data from *channel*, replacing any existing instance data.

MQLONG batchHeartBeat() const ;

Returns the **batch heart-beat**.

ImqBoolean setBatchHeartBeat(const MQLONG *heartbeat* = 0L);

Sets the **batch heart-beat**. This method returns TRUE if successful.

ImqString channelName() const ;

Returns the **channel name**.

ImqBoolean setChannelName(const char * *name* = 0);

Sets the **channel name**. This method returns TRUE if successful.

ImqString connectionName() const ;

Returns the **connection name**.

ImqBoolean setConnectionName(const char * *name* = 0);

Sets the **connection name**. This method returns TRUE if successful.

size_t headerCompressionCount() const ;

Returns the supported header data compression techniques count.

ImqBoolean headerCompression(const size_t *count*, MQLONG *compress* []) const ;

Returns copies of the supported header data compression techniques in **compress**. This method returns TRUE if successful.

ImqBoolean setHeaderCompression(const size_t *count*, const MQLONG *compress* []);

Sets the supported header data compression techniques to **compress**.

Sets the supported header data compression techniques count to **count**.

This method returns TRUE if successful.

MQLONG heartBeatInterval() const ;

Returns the **heart-beat interval**.

ImqBoolean setHeartBeatInterval(const MQLONG *interval* = 300L);

Sets the **heart-beat interval**. This method returns TRUE if successful.

MQLONG keepAliveInterval() const ;

Returns the **keep alive interval**.

ImqBoolean setKeepAliveInterval(const MQLONG *interval* = MQKAI_AUTO);

Sets the **keep alive interval**. This method returns TRUE if successful.

ImqString localAddress() const ;

Returns the **local address**.

ImqBoolean setLocalAddress (const char * *address* = 0);

Sets the **local address**. This method returns TRUE if successful.

MQLONG maximumMessageLength() const ;

Returns the **maximum message length**.

ImqBoolean setMaximumMessageLength(const MQLONG *length* = 4194304L);

Sets the **maximum message length**. This method returns TRUE if successful.

size_t messageCompressionCount() const ;

Returns the supported message data compression techniques count.

ImqBoolean messageCompression(const size_t *count*, MQLONG *compress* []) const ;

Returns copies of the supported message data compression techniques in **compress**. This method returns TRUE if successful.

ImqBoolean setMessageCompression(const size_t *count*, const MQLONG *compress* []);

Sets the supported message data compression techniques to **compress**.

Sets the supported message data compression techniques count to **count**.

This method returns TRUE if successful.

ImqString modeName() const ;

Returns the **mode name**.

ImqBoolean setModeName(const char * *name* = 0);

Sets the **mode name**. This method returns TRUE if successful.

ImqString password() const ;

Returns the **password**.

ImqBoolean setPassword(const char * *password* = 0);

Sets the **password**. This method returns TRUE if successful.

size_t receiveExitCount() const ;

Returns the **receive exit count**.

ImqString receiveExitName();

Returns the first of the **receive exit names**, if any. If the **receive exit count** is zero, it returns an empty string.

ImqBoolean receiveExitNames(const size_t *count*, ImqString * *names* []);

Returns copies of the **receive exit names** in *names*. Sets any *names* in excess of **receive exit count** to null strings. This method returns TRUE if successful.

ImqBoolean setReceiveExitName(const char * *name* = 0);

Sets the **receive exit names** to the single *name*. *name* can be blank or null. Sets the **receive exit count** to either 1 or zero. Clears the **receive user data**. This method returns TRUE if successful.

ImqBoolean setReceiveExitNames(const size_t *count*, const char * *names* []);

Sets the **receive exit names** to *names*. Individual *names* values must not be blank or null. Sets the **receive exit count** to *count*. Clears the **receive user data**. This method returns TRUE if successful.

ImqBoolean setReceiveExitNames(const size_t *count*, const ImqString * *names* []);

Sets the **receive exit names** to *names*. Individual *names* values must not be blank or null. Sets the **receive exit count** to *count*. Clears the **receive user data**. This method returns TRUE if successful.

ImqString receiveUserData();

Returns the first of the **receive user data** items, if any. If the **receive exit count** is zero, returns an empty string.

ImqBoolean receiveUserData(const size_t *count*, ImqString * *data* []);

Returns copies of the **receive user data** items in *data*. Sets any *data* in excess of **receive exit count** to null strings. This method returns TRUE if successful.

ImqBoolean setReceiveUserData(const char * *data* = 0);

Sets the **receive user data** to the single item *data*. If *data* is not null, **receive exit count** must be at least 1. This method returns TRUE if successful.

ImqBoolean setReceiveUserData(const size_t *count*, const char * *data* []);

Sets the **receive user data** to *data*. *count* must be no greater than the **receive exit count**. This method returns TRUE if successful.

ImqBoolean setReceiveUserData(const size_t *count*, const ImqString * *data* []);

Sets the **receive user data** to *data*. *count* must be no greater than the **receive exit count**. This method returns TRUE if successful.

ImqString securityExitName() const ;

Returns the **security exit name**.

ImqBoolean setSecurityExitName(const char * *name* = 0);

Sets the **security exit name**. This method returns TRUE if successful.

ImqString securityUserData() const ;

Returns the **security user data**.

ImqBoolean setSecurityUserData(const char * *data* = 0);

Sets the **security user data**. This method returns TRUE if successful.

size_t sendExitCount() const ;

Returns the **send exit count**.

ImqString sendExitName();

Returns the first of the **send exit names**, if any. Returns an empty string if the **send exit count** is zero.

ImqBoolean sendExitNames(const size_t count, ImqString * names []);

Returns copies of the **send exit names** in *names*. Sets any *names* in excess of **send exit count** to null strings. This method returns TRUE if successful.

ImqBoolean setSendExitName(const char * name = 0);

Sets the **send exit names** to the single *name*. *name* can be blank or null. Sets the **send exit count** to either 1 or zero. Clears the **send user data**. This method returns TRUE if successful

ImqBoolean setSendExitNames(const size_t count, const char * names []);

Sets the **send exit names** to *names*. Individual *names* values must not be blank or null. Sets the **send exit count** to *count*. Clears the **send user data**. This method returns TRUE if successful.

ImqBoolean setSendExitNames(const size_t count, const ImqString * names []);

Sets the **send exit names** to *names*. Individual *names* values must not be blank or null. Sets the **send exit count** to *count*. Clears the **send user data**. This method returns TRUE if successful.

ImqString sendUserData();

Returns the first of the **send user data** items, if any. , Returns an empty string if the **send exit count** is zero.

ImqBoolean sendUserData(const size_t count, ImqString * data []);

Returns copies of the **send user data** items in *data*. Sets any *data* in excess of **send exit count** to null strings. This method returns TRUE if successful.

ImqBoolean setSendUserData(const char * data = 0);

Sets the **send user data** to the single item *data*. If *data* is not null, **send exit count** must be at least 1. This method returns TRUE if successful.

ImqBoolean setSendUserData(const size_t count, const char * data []);

Sets the **send user data** to *data*. *count* must be no greater than the **send exit count**. This method returns TRUE if successful.

ImqBoolean setSendUserData(const size_t count, const ImqString * data []);

Sets the **send user data** to *data*. *count* must be no greater than the **send exit count**. This method returns TRUE if successful.

ImqString sslCipherSpecification() const ;

Returns the SSL cipher specification.

ImqBoolean setSslCipherSpecification(const char * name = 0);

Sets the SSL cipher specification. This method returns TRUE if successful.

MQLONG sslClientAuthentication() const ;

Returns the SSL client authentication type.

ImqBoolean setSslClientAuthentication(const MQLONG auth = MQSCA_REQUIRED);

Sets the SSL client authentication type. This method returns TRUE if successful.

ImqString sslPeerName() const ;

Returns the SSL peer name.

ImqBoolean setSslPeerName(const char * name = 0);

Sets the SSL peer name. This method returns TRUE if successful.

ImqString transactionProgramName() const ;

Returns the **transaction program name**.

ImqBoolean setTransactionProgramName(const char * name = 0);

Sets the **transaction program name**. This method returns TRUE if successful.

MQLONG transportType() const ;

Returns the **transport type**.

ImqBoolean setTransportType(const MQLONG type = MQXPT_LU62);

Sets the **transport type**. This method returns TRUE if successful.

ImqString userId() const ;

Returns the **user id**.

ImqBoolean setId(const char * id = 0);

Sets the **user id**. This method returns TRUE if successful.

Reason codes

- MQRC_DATA_LENGTH_ERROR
- MQRC_ITEM_COUNT_ERROR
- MQRC_NULL_POINTER
- MQRC_SOURCE_BUFFER_ERROR

ImqCICSBridgeHeader C++ class

This class encapsulates specific features of the MQCIH data structure.

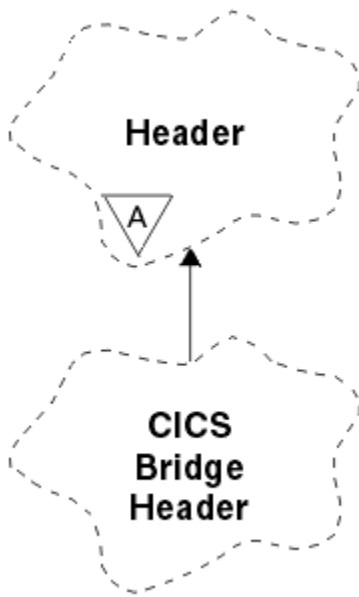


Figure 50. *ImqCICSBridgeHeader* class

Objects of this class are used by applications that send messages to the CICS bridge through IBM MQ for z/OS.

- [“Object attributes” on page 1849](#)
- [“Constructors” on page 1852](#)
- [“Overloaded ImqItem methods” on page 1852](#)
- [“Object methods \(public\)” on page 1852](#)
- [“Object data \(protected\)” on page 1854](#)
- [“Reason codes” on page 1854](#)
- [“Return codes” on page 1854](#)

Object attributes

ADS descriptor

Send/receive ADS descriptor. This is set using MQCADSD_NONE. The initial value is MQCADSD_NONE. The following additional values are possible:

- MQCADSD_NONE
- MQCADSD_SEND
- MQCADSD_RECV

- MQCADSD_MSGFORMAT

attention identifier

AID key. The field must be of length MQ_ATTENTION_ID_LENGTH.

authenticator

RACF password or passticket. The initial value contains blanks, of length MQ_AUTHENTICATOR_LENGTH.

bridge abend code

Bridge abend code, of length MQ_ABEND_CODE_LENGTH. The initial value is four blank characters. The value returned in this field is dependent on the return code. See [Table 676 on page 1854](#) for more details.

bridge cancel code

Bridge abend transaction code. The field is reserved, must contain blanks, and be of length MQ_CANCEL_CODE_LENGTH.

bridge completion code

Completion code, which can contain either the IBM MQ completion code or the CICS EIBRESP value. The field has the initial value of MQCC_OK. The value returned in this field is dependent on the return code. See [Table 676 on page 1854](#) for more details.

bridge error offset

Bridge error offset. The initial value is zero. This attribute is read-only.

bridge reason code

Reason code. This field can contain either the IBM MQ reason or the CICS EIBRESP2 value. The field has the initial value of MQRC_NONE. The value returned in this field is dependent on the return code. See [Table 676 on page 1854](#) for more details.

bridge return code

Return code from the CICS bridge. The initial value is MQCRC_OK.

conversational task

Whether the task can be conversational. The initial value is MQCCT_NO. The following additional values are possible:

- MQCCT_YES
- MQCCT_NO

cursor position

Cursor position. The initial value is zero.

facility keep time

CICS bridge facility release time.

facility like

Terminal emulated attribute. The field must be of length MQ_FACILITY_LIKE_LENGTH.

facility token

BVT token value. The field must be of length MQ_FACILITY_LENGTH. The initial value is MQCFAC_NONE.

function

Function, which can contain either the IBM MQ call name or the CICS EIBFN function. The field has the initial value of MQCFUNC_NONE, with length MQ_FUNCTION_LENGTH. The value returned in this field is dependent on the return code. See [Table 676 on page 1854](#) for more details.

The following additional values are possible when **function** contains an IBM MQ call name:

- MQCFUNC_MQCONN
- MQCFUNC_MQGET
- MQCFUNC_MQINQ
- MQCFUNC_NONE
- MQCFUNC_MQOPEN

- MQCFUNC_PUT
- MQCFUNC_MQPUT1

get wait interval

Wait interval for an MQGET call issued by the CICS bridge task. The initial value is MQCGWI_DEFAULT. The field applies only when **uow control** has the value MQCUOWC_FIRST. The following additional values are possible:

- MQCGWI_DEFAULT
- MQWI_UNLIMITED

link type

Link type. The initial value is MQCLT_PROGRAM. The following additional values are possible:

- MQCLT_PROGRAM
- MQCLT_TRANSACTION

next transaction identifier

ID of the next transaction to attach. The field must be of length MQ_TRANSACTION_ID_LENGTH.

output data length

COMMAREA data length. The initial value is MQCODL_AS_INPUT.

reply-to format

Format name of the reply message. The initial value is MQFMT_NONE with length MQ_FORMAT_LENGTH.

start code

Transaction start code. The field must be of length MQ_START_CODE_LENGTH. The initial value is MQCSC_NONE. The following additional values are possible:

- MQCSC_START
- MQCSC_STARTDATA
- MQCSC_TERMINPUT
- MQCSC_NONE

task end status

Task end status. The initial value is MQCTES_NOSYNC. The following additional values are possible:

- MQCTES_COMMIT
- MQCTES_BACKOUT
- MQCTES_ENDTASK
- MQCTES_NOSYNC

transaction identifier

ID of the transaction to attach. The initial value must contain blanks, and must be of length MQ_TRANSACTION_ID_LENGTH. The field applies only when **uow control** has the value MQCUOWC_FIRST or MQCUOWC_ONLY.

UOW control

UOW control. The initial value is MQCUOWC_ONLY. The following additional values are possible:

- MQCUOWC_FIRST
- MQCUOWC_MIDDLE
- MQCUOWC_LAST
- MQCUOWC_ONLY
- MQCUOWC_COMMIT
- MQCUOWC_BACKOUT
- MQCUOWC_CONTINUE

version

The MQCIH version number. The initial value is MQCIH_VERSION_2. The only other supported value is MQCIH_VERSION_1.

Constructors

ImqCICSBridgeHeader();

The default constructor.

ImqCICSBridgeHeader(const ImqCICSBridgeHeader & header);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut(ImqMessage & msg);

Inserts an MQCIH data structure into the message buffer at the beginning, moving existing message data further along, and sets the message format to MQFMT_CICS.

See the parent class method description for more details.

virtual ImqBoolean pasteIn(ImqMessage & msg);

Reads an MQCIH data structure from the message buffer. To be successful, the encoding of the *msg* object must be MQENC_NATIVE. Retrieve messages with MQGMO_CONVERT to MQENC_NATIVE. To be successful, the ImqMessage format must be MQFMT_CICS.

See the parent class method description for more details.

Object methods (public)

void operator = (const ImqCICSBridgeHeader & header);

Copies instance data from the *header*, replacing the existing instance data.

MQLONG ADSDescriptor() const;

Returns a copy of the **ADS descriptor**.

void setADSDescriptor(const MQLONG descriptor = MQCADSD_NONE);

Sets the **ADS descriptor**.

ImqString attentionIdentifier() const;

Returns a copy of the **attention identifier**, padded with trailing blanks to length MQ_ATTENTION_ID_LENGTH.

void setAttentionIdentifier(const char * data = 0);

Sets the **attention identifier**, padded with trailing blanks to length MQ_ATTENTION_ID_LENGTH. If no *data* is supplied, resets **attention identifier** to the initial value.

ImqString authenticator() const;

Returns a copy of the **authenticator**, padded with trailing blanks to length MQ_AUTHENTICATOR_LENGTH.

void setAuthenticator(const char * data = 0);

Sets the **authenticator**, padded with trailing blanks to length MQ_AUTHENTICATOR_LENGTH. If no *data* is supplied, resets **authenticator** to the initial value.

ImqString bridgeAbendCode() const;

Returns a copy of the **bridge abend code**, padded with trailing blanks to length MQ_ABEND_CODE_LENGTH.

ImqString bridgeCancelCode() const;

Returns a copy of the **bridge cancel code**, padded with trailing blanks to length MQ_CANCEL_CODE_LENGTH.

void setBridgeCancelCode(const char * data = 0);

Sets the **bridge cancel code**, padded with trailing blanks to length MQ_CANCEL_CODE_LENGTH. If no *data* is supplied, resets the **bridge cancel code** to the initial value.

MQLONG bridgeCompletionCode() const;
Returns a copy of the **bridge completion code**.

MQLONG bridgeErrorOffset() const ;
Returns a copy of the **bridge error offset**.

MQLONG bridgeReasonCode() const;
Returns a copy of the **bridge reason code**.

MQLONG bridgeReturnCode() const;
Returns the **bridge return code**.

MQLONG conversationalTask() const;
Returns a copy of the **conversational task**.

void setConversationalTask(const MQLONG task = MQCCT_NO);
Sets the **conversational task**.

MQLONG cursorPosition() const ;
Returns a copy of the **cursor position**.

void setCursorPosition(const MQLONG position = 0);
Sets the **cursor position**.

MQLONG facilityKeepTime() const;
Returns a copy of the **facility keep time**.

void setFacilityKeepTime(const MQLONG time = 0);
Sets the **facility keep time**.

ImqString facilityLike() const;
Returns a copy of the **facility like**, padded with trailing blanks to length MQ_FACILITY_LIKE_LENGTH.

void setFacilityLike(const char * name = 0);
Sets the **facility like**, padded with trailing blanks to length MQ_FACILITY_LIKE_LENGTH. If no *name* is supplied, resets **facility like** the initial value.

ImqBinary facilityToken() const;
Returns a copy of the **facility token**.

ImqBoolean setFacilityToken(const ImqBinary & token);
Sets the **facility token**. The **data length** of *token* must be either zero or MQ_FACILITY_LENGTH. It returns TRUE if successful.

void setFacilityToken(const MQBYTE8 token = 0);
Sets the **facility token**. *token* can be zero, which is the same as specifying MQCFAC_NONE. If *token* is nonzero it must address MQ_FACILITY_LENGTH bytes of binary data. When using predefined values such as MQCFAC_NONE, you might need to make a cast to ensure a signature match. For example, (MQBYTE *)MQCFAC_NONE.

ImqString function() const;
Returns a copy of the **function**, padded with trailing blanks to length MQ_FUNCTION_LENGTH.

MQLONG getWaitInterval() const;
Returns a copy of the **get wait interval**.

void setGetWaitInterval(const MQLONG interval = MQCGWI_DEFA
Sets the **get wait interval**.

MQLONG linkType() const;
Returns a copy of the **link type**.

void setLinkType(const MQLONG type = MQCLT_PROGRAM);
Sets the **link type**.

ImqString nextTransactionIdentifier() const ;
Returns a copy of the **next transaction identifier** data, padded with trailing blanks to length MQ_TRANSACTION_ID_LENGTH.

MQLONG outputDataLength() const;
Returns a copy of the **output data length**.

void setOutputDataLength(const MQLONG length = MQCODL_AS_INPUT);

Sets the **output data length**.

ImqString replyToFormat() const;

Returns a copy of the **reply-to format** name, padded with trailing blanks to length MQ_FORMAT_LENGTH.

void setReplyToFormat(const char * name = 0);

Sets the **reply-to format**, padded with trailing blanks to length MQ_FORMAT_LENGTH. If no *name* is supplied, resets **reply-to format** to the initial value.

ImqString startCode() const;

Returns a copy of the **start code**, padded with trailing blanks to length MQ_START_CODE_LENGTH.

void setStartCode(const char * data = 0);

Sets the **start code** data, padded with trailing blanks to length MQ_START_CODE_LENGTH. If no *data* is supplied, resets **start code** to the initial value.

MQLONG taskEndStatus() const;

Returns a copy of the **task end status**.

ImqString transactionIdentifier() const;

Returns a copy of the **transaction identifier** data, padded with trailing blanks to the length MQ_TRANSACTION_ID_LENGTH.

void setTransactionIdentifier(const char * data = 0);

Sets the **transaction identifier**, padded with trailing blanks to length MQ_TRANSACTION_ID_LENGTH. If no *data* is supplied, resets **transaction identifier** to the initial value.

MQLONG UOWControl() const;

Returns a copy of the **UOW control**.

void setUOWControl(const MQLONG control = MQCUOWC_ONLY);

Sets the **UOW control**.

MQLONG version() const;

Returns the **version** number.

ImqBoolean setVersion(const MQLONG version = MQCIH_VERSION_2);

Sets the **version** number. It returns TRUE if successful.

Object data (protected)

MQLONG olVersion

The maximum MQCIH version number that can be accommodated in the storage allocated for *opcih*.

PMQCIH opcih

The address of an MQCIH data structure. The amount of storage allocated is indicated by *olVersion*.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR
- MQRC_WRONG_VERSION

Return codes

Table 676. ImqCICSBridgeHeader class return codes				
Return Code	Function	CompCode	Reason	Abend Code
MQCRC_OK				
MQCRC_BRIDGE_ERROR			MQFB_CICS	

Table 676. ImqCICSBridgeHeader class return codes (continued)				
Return Code	Function	CompCode	Reason	Abend Code
MQCRC_MQ_API_ERROR	IBM MQ call name	IBM MQ CompCode	IBM MQ Reason	
MQCRC_BRIDGE_TIMEOUT	IBM MQ call name	IBM MQ CompCode	IBM MQ Reason	
MQCRC_CICS_EXEC_ERROR	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_SECURITY_ERROR	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_PROGRAM_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_TRANSID_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_BRIDGE_ABEND				CICS ABCODE
MQCRC_APPLICATION_ABEND				CICS ABCODE

ImqDeadLetterHeader C++ class

This class encapsulates features of the MQDLH data structure.

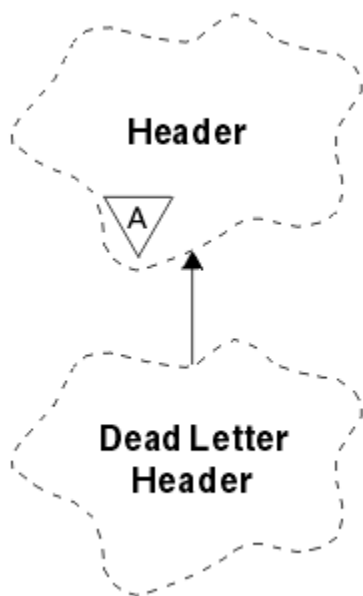


Figure 51. ImqDeadLetterHeader class

Objects of this class are typically used by an application that encounters a message that cannot be processed. A new message comprising a dead-letter header and the message content is placed on the dead-letter queue, and the message is discarded.

- [“Object attributes” on page 1856](#)
- [“Constructors” on page 1856](#)
- [“Overloaded ImqItem methods” on page 1856](#)
- [“Object methods \(public\)” on page 1856](#)
- [“Object data \(protected\)” on page 1857](#)
- [“Reason codes” on page 1857](#)

Object attributes

dead-letter reason code

The reason the message arrived on the dead-letter queue. The initial value is MQRC_NONE.

destination queue manager name

The name of the original destination queue manager. The name is a string of length MQ_Q_MGR_NAME_LENGTH. Its initial value is null.

destination queue name

The name of the original destination queue. The name is a string of length MQ_Q_NAME_LENGTH. Its initial value is null.

put application name

The name of the application that put the message on the dead-letter queue. The name is a string of length MQ_PUT_APPL_NAME_LENGTH. Its initial value is null.

put application type

The type of application that put the message on the dead-letter queue. The initial value is zero.

put date

The date when the message was put on the dead-letter queue. The date is a string of length MQ_PUT_DATE_LENGTH. Its initial value is a null string.

put time

The time when the message was put on the dead-letter queue. The time is a string of length MQ_PUT_TIME_LENGTH. Its initial value is a null string.

Constructors

ImqDeadLetterHeader();

The default constructor.

ImqDeadLetterHeader(const ImqDeadLetterHeader & header);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & msg);

Inserts an MQDLH data structure into the message buffer at the beginning, moving existing message data further along. Sets the *msg format* to MQFMT_DEAD_LETTER_HEADER.

See the ImqHeader class method description on page [“ImqHeader C++ class” on page 1864](#) for further details.

virtual ImqBoolean pasteIn (ImqMessage & msg);

Reads an MQDLH data structure from the message buffer.

To be successful, the ImqMessage *format* must be MQFMT_DEAD_LETTER_HEADER.

See the ImqHeader class method description on page [“ImqHeader C++ class” on page 1864](#) for further details.

Object methods (public)

void operator = (const ImqDeadLetterHeader & header);

Copies instance data is copied from *header*, replacing the existing instance data.

MQLONG deadLetterReasonCode () const ;

Returns the **dead-letter reason code**.

void setDeadLetterReasonCode (const MQLONG reason);

Sets the **dead-letter reason code**.

ImqString destinationQueueManagerName () const ;

Returns the **destination queue manager name**, stripped of any trailing blanks.

void setDestinationQueueManagerName (const char * *name*);

Sets the **destination queue manager name**. Truncates data longer than MQ_Q_MGR_NAME_LENGTH (48 characters).

ImqString destinationQueueName () const ;

Returns a copy of the **destination queue name**, stripped of any trailing blanks.

void setDestinationQueueName (const char * *name*);

Sets the **destination queue name**. Truncates data longer than MQ_Q_NAME_LENGTH (48 characters).

ImqString putApplicationName () const ;

Returns a copy of the **put application name**, stripped of any trailing blanks.

void setPutApplicationName (const char * *name* = 0);

Sets the **put application name**. Truncates data longer than MQ_PUT_APPL_NAME_LENGTH (28 characters).

MQLONG putApplicationType () const ;

Returns the **put application type**.

void setPutApplicationType (const MQLONG *type* = MQAT_NO_CONTEXT);

Sets the **put application type**.

ImqString putDate () const ;

Returns a copy of the **put date**, stripped of any trailing blanks.

void setPutDate (const char * *date* = 0);

Sets the **put date**. Truncates data longer than MQ_PUT_DATE_LENGTH (8 characters).

ImqString putTime () const ;

Returns a copy of the **put time**, stripped of any trailing blanks.

void setPutTime (const char * *time* = 0);

Sets the **put time**. Truncates data longer than MQ_PUT_TIME_LENGTH (8 characters).

Object data (protected)

MQDLH *omqdlh*

The MQDLH data structure.

Reason codes

- MQRC_INCONSISTENT_FORMAT
- MQRC_STRUC_ID_ERROR
- MQRC_ENCODING_ERROR

ImqDistributionList C++ class

This class encapsulates a dynamic distribution list that references one or more queues for the purpose of sending a message or messages to multiple destinations.

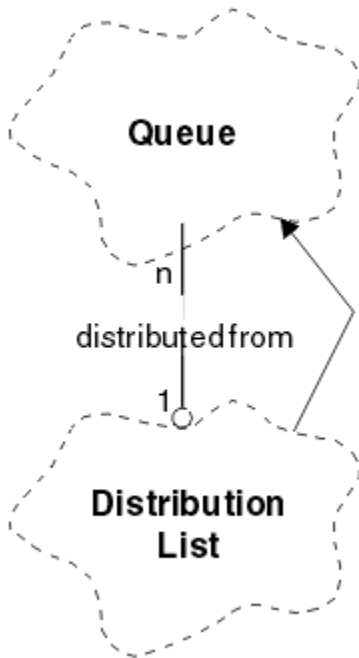


Figure 52. ImqDistributionList class

- [“Object attributes” on page 1858](#)
- [“Constructors” on page 1858](#)
- [“Object methods \(public\)” on page 1858](#)
- [“Object methods \(protected\)” on page 1859](#)

Object attributes

first distributed queue

The first of one or more objects of class , in no particular order, in which the **distribution list reference** addresses this object.

Initially there are no such objects. To open an ImqDistributionList successfully, there must be at least one such object.

Note: When an ImqDistributionList object is opened, any open objects that reference it are automatically closed.

Constructors

ImqDistributionList();

The default constructor.

ImqDistributionList(const ImqDistributionList & list);

The copy constructor.

Object methods (public)

void operator = (const ImqDistributionList & list);

All objects that reference **this** object are dereferenced before copying. No objects will reference **this** object after the invocation of this method.

*** firstDistributedQueue () const ;**
Returns the **first distributed queue**.

Object methods (protected)

void setFirstDistributedQueue (* queue = 0);
Sets the **first distributed queue**.

ImqError C++ class

This abstract class provides information on errors associated with an object.

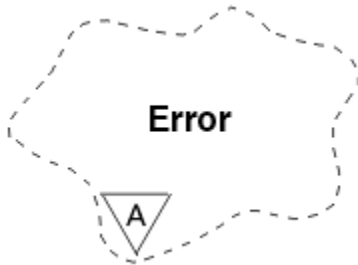


Figure 53. *ImqError* class

- [“Object attributes” on page 1859](#)
- [“Constructors” on page 1859](#)
- [“Object methods \(public\)” on page 1859](#)
- [“Object methods \(protected\)” on page 1860](#)
- [“Reason codes” on page 1860](#)

Object attributes

completion code

The most recent completion code. The initial value is zero. The following additional values are possible:

- MQCC_OK
- MQCC_WARNING
- MQCC_FAILED

reason code

The most recent reason code. The initial value is zero.

Constructors

ImqError ();

The default constructor.

ImqError (const ImqError & error);

The copy constructor.

Object methods (public)

void operator = (const ImqError & error);

Copies instance data from *error*, replacing the existing instance data.

void clearErrorCodes ();

Sets the **completion code** and **reason code** both to zero.

MQLONG completionCode () const ;

Returns the **completion code**.

MQLONG reasonCode () const ;

Returns the **reason code**.

Object methods (protected)

ImqBoolean checkReadPointer (const void * *pointer*, const size_t *length*);

Verifies that the combination of pointer and length is valid for read-only access, and returns TRUE if successful.

ImqBoolean checkWritePointer (const void * *pointer*, const size_t *length*);

Verifies that the combination of pointer and length is valid for read-write access, and returns TRUE if successful.

void setCompletionCode (const MQLONG *code* = 0);

Sets the **completion code**.

void setReasonCode (const MQLONG *code* = 0);

Sets the **reason code**.

Reason codes

- MQRC_BUFFER_ERROR

ImqGetMessageOptions C++ class

This class encapsulates the MQGMO data structure

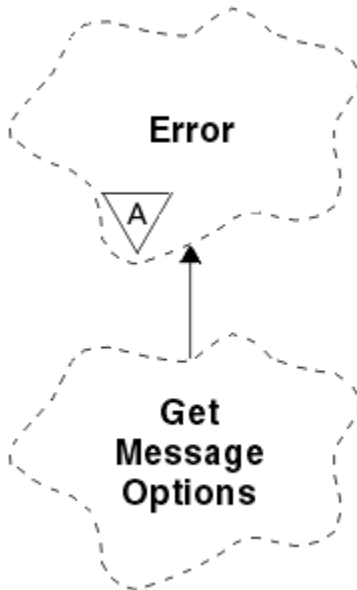


Figure 54. *ImqGetMessageOptions* class

- [“Object attributes” on page 1861](#)
- [“Constructors” on page 1862](#)
- [“Object methods \(public\)” on page 1862](#)
- [“Object methods \(protected\)” on page 1863](#)
- [“Object data \(protected\)” on page 1863](#)
- [“Reason codes” on page 1863](#)

Object attributes

group status

Status of a message for a group of messages. The initial value is MQGS_NOT_IN_GROUP. The following additional values are possible:

- MQGS_MSG_IN_GROUP
- MQGS_LAST_MSG_IN_GROUP

match options

Options for selecting incoming messages. The initial value is MQMO_MATCH_MSG_ID | MQMO_MATCH_CORREL_ID. The following additional values are possible:

- MQMO_GROUP_ID
- MQMO_MATCH_MSG_SEQ_NUMBER
- MQMO_MATCH_OFFSET
- MQMO_MSG_TOKEN
- MQMO_NONE

message token

Message token. A binary value (MQBYTE16) of length MQ_MSG_TOKEN_LENGTH. The initial value is MQMTOK_NONE.

options

Options applicable to a message. The initial value is MQGMO_NO_WAIT. The following additional values are possible:

- MQGMO_WAIT
- MQGMO_SYNCPOINT
- MQGMO_SYNCPOINT_IF_PERSISTENT
- MQGMO_NO_SYNCPOINT
- MQGMO_MARK_SKIP_BACKOUT
- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_NEXT
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_MSG_UNDER_CURSOR
- MQGMO_LOCK
- MQGMO_UNLOCK
- MQGMO_ACCEPT_TRUNCATED_MSG
- MQGMO_SET_SIGNAL
- MQGMO_FAIL_IF QUIESCING
- MQGMO_CONVERT
- MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MSG
- MQGMO_ALL_MSGS_AVAILABLE
- MQGMO_ALL_SEGMENTS_AVAILABLE
- MQGMO_NONE

resolved queue name

Resolved queue name. This attribute is read-only. Names are never longer than 48 characters and can be padded to that length with nulls. The initial value is a null string.

returned length

Returned length. The initial value is MQRL_UNDEFINED. This attribute is read-only.

segmentation

The ability to segment a message. The initial value is MQSEG_INHIBITED. The additional value, MQSEG_ALLOWED, is possible.

segment status

The segmentation status of a message. The initial value is MQSS_NOT_A_SEGMENT. The following additional values are possible:

- MQSS_SEGMENT
- MQSS_LAST_SEGMENT

syncpoint participation

TRUE when messages are retrieved under syncpoint control.

wait interval

The length of time that the class **get** method pauses while waiting for a suitable message to arrive, if one is not already available. The initial value is zero, which effects an indefinite wait. The additional value, MQWI_UNLIMITED, is possible. This attribute is ignored unless the **options** include MQGMO_WAIT.

Constructors

ImqGetMessageOptions();

The default constructor.

ImqGetMessageOptions(const ImqGetMessageOptions & gmo);

The copy constructor.

Object methods (public)

void operator = (const ImqGetMessageOptions & gmo);

Copies instance data from *gmo*, replacing the existing instance data.

MQCHAR groupStatus () const ;

Returns the **group status**.

void setGroupStatus (const MQCHAR status);

Sets the **group status**.

MLONG matchOptions () const ;

Returns the **match options**.

void setMatchOptions (const MLONG options);

Sets the **match options**.

ImqBinary messageToken () const;

Returns the **message token**.

ImqBoolean setMessageToken(const ImqBinary & token);

Sets the **message token**. The **data length** of *token* must be either zero or MQ_MSG_TOKEN_LENGTH. This method returns TRUE if successful.

void setMessageToken(const MQBYTE16 token = 0);

Sets the **message token**. *token* can be zero, which is the same as specifying MQMTOK_NONE. If *token* is nonzero, then it must address MQ_MSG_TOKEN_LENGTH bytes of binary data.

When using predefined values, such as MQMTOK_NONE, you might not need to make a cast to ensure a signature match, for example (MQBYTE *)MQMTOK_NONE.

MLONG options () const ;

Returns the **options**.

void setOptions (const MLONG options);

Sets the **options**, including the **syncpoint participation** value.

ImqString resolvedQueueName () const ;

Returns a copy of the **resolved queue name**.

MQLONG returnedLength() const;

Returns the **returned length**.

MQCHAR segmentation () const ;

Returns the **segmentation**.

void setSegmentation (const MQCHAR value);

Sets the **segmentation**.

MQCHAR segmentStatus () const ;

Returns the **segment status**.

void setSegmentStatus (const MQCHAR status);

Sets the **segment status**.

ImqBoolean syncPointParticipation () const ;

Returns the **syncpoint participation** value, which is TRUE if the **options** include either MQGMO_SYNCPOINT or MQGMO_SYNCPOINT_IF_PERSISTENT.

void setSyncPointParticipation (const ImqBoolean sync);

Sets the **syncpoint participation** value. If *sync* is TRUE, alters the **options** to include MQGMO_SYNCPOINT, and to exclude both MQGMO_NO_SYNCPOINT and MQGMO_SYNCPOINT_IF_PERSISTENT. If *sync* is FALSE, alters the **options** to include MQGMO_NO_SYNCPOINT, and to exclude both MQGMO_SYNCPOINT and MQGMO_SYNCPOINT_IF_PERSISTENT.

MQLONG waitInterval () const ;

Returns the **wait interval**.

void setWaitInterval (const MQLONG interval);

Sets the **wait interval**.

Object methods (protected)

static void setVersionSupported (const MQLONG);

Sets the **MQGMO** version. Defaults to **MQGMO_VERSION_3**.

Object data (protected)

MQGMO omqgmo

An MQGMO Version 2 data structure. Access MQGMO fields supported for MQGMO_VERSION_2 only.

PMQGMO opgmo

The address of an MQGMO data structure. The version number for this address is indicated in *olVersion*. Inspect the version number before accessing MQGMO fields, to ensure that they are present.

MQLONG olVersion

The version number of the MQGMO data structure addressed by *opgmo*.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR

ImqHeader C++ class

This abstract class encapsulates common features of the MQDLH data structure.

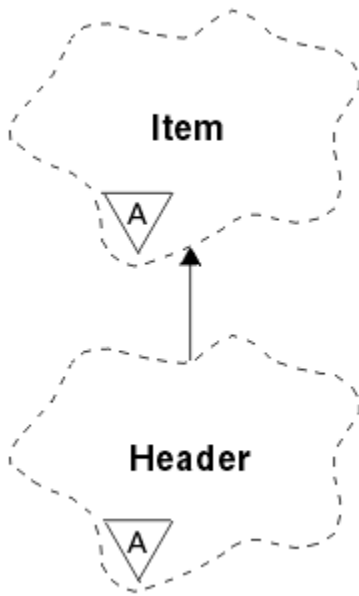


Figure 55. ImqHeader class

- [“Object attributes” on page 1864](#)
- [“Constructors” on page 1864](#)
- [“Object methods \(public\)” on page 1865](#)

Object attributes

character set

The original coded character set identifier. Initially MQCCSI_Q_MGR.

encoding

The original encoding. Initially MQENC_NATIVE.

format

The original format. Initially MQFMT_NONE.

header flags

The initial values are:

- Zero for objects of the ImqDeadLetterHeader class
- MQIIH_NONE for objects of the ImqIMSBridgeHeader class
- MQRMHF_LAST for objects of the ImqReferenceHeader class
- MQCIH_NONE for objects of the ImqCICSBridgeHeader class
- MQWIIH_NONE for objects of the ImqWorkHeader class

Constructors

ImqHeader();

The default constructor.

ImqHeader(const ImqHeader & header);

The copy constructor.

Object methods (public)

void operator = (const ImqHeader & *header*);

Copies instance data from *header*, replacing the existing instance data.

virtual MQLONG characterSet () const ;

Returns the **character set**.

virtual void setCharacterSet (const MQLONG *ccsid* = MQCCSI_Q_MGR);

Sets the **character set**.

virtual MQLONG encoding () const ;

Returns the **encoding**.

virtual void setEncoding (const MQLONG *encoding* = MQENC_NATIVE);

Sets the **encoding**.

virtual ImqString format () const ;

Returns a copy of the **format**, including trailing blanks.

virtual void setFormat (const char * *name* = 0);

Sets the **format**, padded to 8 characters with trailing blanks.

virtual MQLONG headerFlags () const ;

Returns the **header flags**.

virtual void setHeaderFlags (const MQLONG *flags* = 0);

Sets the **header flags**.

ImqIMSBridgeHeader C++ class

This class encapsulates features of the MQIIH data structure.

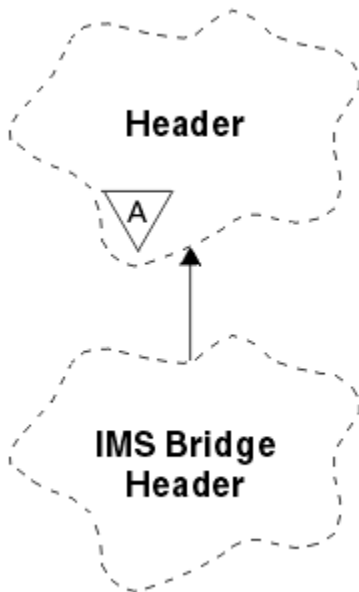


Figure 56. *ImqIMSBridgeHeader* class

Objects of this class are used by applications that send messages to the IMS bridge through IBM MQ for z/OS.

Note: The *ImqHeader* **character set** and **encoding** must have default values and must not be set to any other values.

- [“Object attributes” on page 1866](#)
- [“Constructors” on page 1866](#)
- [“Overloaded *ImqItem* methods” on page 1866](#)

- [“Object methods \(public\)” on page 1867](#)
- [“Object data \(protected\)” on page 1867](#)
- [“Reason codes” on page 1867](#)

Object attributes

authenticator

RACF password or passticket, of length MQ_AUTHENTICATOR_LENGTH. The initial value is MQIAUT_NONE.

commit mode

Commit mode. See the *OTMA User's Guide* for more information about IMS commit modes. The initial value is MQICM_COMMIT_THEN_SEND. The additional value, MQICM_SEND_THEN_COMMIT, is possible.

logical terminal override

Logical terminal override, of length MQ_LTERM_OVERRIDE_LENGTH. The initial value is a null string.

message format services map name

MFS map name, of length MQ_MFS_MAP_NAME_LENGTH. The initial value is a null string.

reply-to format

Format of any reply, of length MQ_FORMAT_LENGTH. The initial value is MQFMT_NONE.

security scope

Scope of IMS security processing. The initial value is MQISS_CHECK. The additional value, MQISS_FULL, is possible.

transaction instance id

Transaction instance identity, a binary (MQBYTE16) value of length MQ_TRAN_INSTANCE_ID_LENGTH. The initial value is MQITII_NONE.

transaction state

State of the IMS conversation. The initial value is MQITS_NOT_IN_CONVERSATION. The additional value, MQITS_IN_CONVERSATION, is possible.

Constructors

ImqIMSBridgeHeader();

The default constructor.

ImqIMSBridgeHeader(const ImqIMSBridgeHeader & header);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & msg);

Inserts an MQIIH data structure into the message buffer at the beginning, moving existing message data further along. Sets the *msg* **format** to MQFMT_IMS.

See the parent class method description for further details.

virtual ImqBoolean pasteIn (ImqMessage & msg);

Reads an MQIIH data structure from the message buffer.

To be successful, the **encoding** of the *msg* object must be MQENC_NATIVE. Retrieve messages with MQGMO_CONVERT to MQENC_NATIVE.

To be successful, the ImqMessage **format** must be MQFMT_IMS.

See the parent class method description for further details.

Object methods (public)

void operator = (const ImqIMSBridgeHeader & *header*);

Copies instance data from *header*, replacing the existing instance data.

ImqString authenticator () const ;

Returns a copy of the **authenticator**, padded with trailing blanks to length MQ_AUTHENTICATOR_LENGTH.

void setAuthenticator (const char * *name*);

Sets the **authenticator**.

MQCHAR commitMode () const ;

Returns the **commit mode**.

void setCommitMode (const MQCHAR *mode*);

Sets the **commit mode**.

ImqString logicalTerminalOverride () const ;

Returns a copy of the **logical terminal override**.

void setLogicalTerminalOverride (const char * *override*);

Sets the **logical terminal override**.

ImqString messageFormatServicesMapName () const ;

Returns a copy of the **message format services map name**.

void setMessageFormatServicesMapName (const char * *name*);

Sets the **message format services map name**.

ImqString replyToFormat () const ;

Returns a copy of the **reply-to format**, padded with trailing blanks to length MQ_FORMAT_LENGTH.

void setReplyToFormat (const char * *format*);

Sets the **reply-to format**, padded with trailing blanks to length MQ_FORMAT_LENGTH.

MQCHAR securityScope () const ;

Returns the **security scope**.

void setSecurityScope (const MQCHAR *scope*);

Sets the **security scope**.

ImqBinary transactionInstanceId () const ;

Returns a copy of the **transaction instance id**.

ImqBoolean setTransactionInstanceId (const ImqBinary & *id*);

Sets the **transaction instance id**. The **data length** of *token* must be either zero or MQ_TRAN_INSTANCE_ID_LENGTH. This method returns TRUE if successful.

void setTransactionInstanceId (const MQBYTE16 *id* = 0);

Sets the **transaction instance id**. *id* can be zero, which is the same as specifying MQITII_NONE. If *id* is nonzero, it must address MQ_TRAN_INSTANCE_ID_LENGTH bytes of binary data. When using predefined values such as MQITII_NONE, you might need to make a cast to ensure a signature match, for example (MQBYTE *)MQITII_NONE.

MQCHAR transactionState () const ;

Returns the **transaction state**.

void setTransactionState (const MQCHAR *state*);

Sets the **transaction state**.

Object data (protected)

MQIIH omqiih

The MQIIH data structure.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR

- MQRC_INCONSISTENT_FORMAT
- MQRC_ENCODING_ERROR
- MQRC_STRUC_ID_ERROR

ImqItem C++ class

This abstract class represents an item, perhaps one of several, within a message.

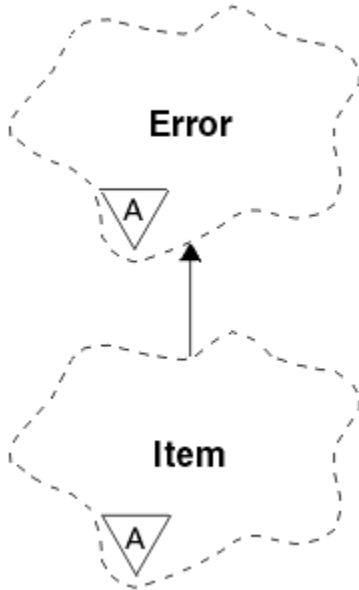


Figure 57. *ImqItem* class

Items are concatenated together in a message buffer. Each specialization is associated with a particular data structure that begins with a structure id.

Polymorphic methods in this abstract class allow items to be copied to and from messages. The `ImqMessage` class **readItem** and **writeItem** methods provide another style of invoking these polymorphic methods that is more natural for application programs.

- [“Object attributes” on page 1868](#)
- [“Constructors” on page 1868](#)
- [“Class methods \(public\)” on page 1869](#)
- [“Object methods \(public\)” on page 1869](#)
- [“Reason codes” on page 1869](#)

Object attributes

structure id

A string of four characters at the beginning of the data structure. This attribute is read-only. Consider this attribute for derived classes. It is not included automatically.

Constructors

ImqItem();

The default constructor.

ImqItem(const ImqItem & item);

The copy constructor.

Class methods (public)

static ImqBoolean structureIdIs (const char * *structure-id-to-test*, const ImqMessage & *msg*);

Returns TRUE if the **structure id** of the next ImqItem in the incoming *msg* is the same as *structure-id-to-test*. The next item is identified as that part of the message buffer currently addressed by the ImqCache **data pointer**. This method relies on the **structure id** and therefore is not guaranteed to work for all ImqItem derived classes.

Object methods (public)

void operator = (const ImqItem & *item*);

Copies instance data from *item*, replacing the existing instance data.

virtual ImqBoolean copyOut (ImqMessage & *msg*) = 0 ;

Writes this object as the next item in an outgoing message buffer, appending it to any existing items. If the write operation is successful, increases the ImqCache **data length**. This method returns TRUE if successful.

Override this method to work with a specific subclass.

virtual ImqBoolean pasteIn (ImqMessage & *msg*) = 0 ;

Reads this object *destructively* from the incoming message buffer. The read is destructive in that the ImqCache **data pointer** is moved on. However, the buffer content remains the same, so data can be re-read by resetting the ImqCache **data pointer**.

The (sub)class of this object must be consistent with the **structure id** found next in the message buffer of the *msg* object.

The **encoding** of the *msg* object should be MQENC_NATIVE. It is recommended that messages be retrieved with the ImqMessage **encoding** set to MQENC_NATIVE, and with the ImqGetMessageOptions **options** including MQGMO_CONVERT.

If the read operation is successful, the ImqCache **data length** is reduced. This method returns TRUE if successful.

Override this method to work with a specific subclass.

Reason codes

- MQRC_ENCODING_ERROR
- MQRC_STRUC_ID_ERROR
- MQRC_INCONSISTENT_FORMAT
- MQRC_INSUFFICIENT_BUFFER
- MQRC_INSUFFICIENT_DATA

ImqMessage C++ class

This class encapsulates an MQMD data structure and also handles the construction and reconstruction of message data.

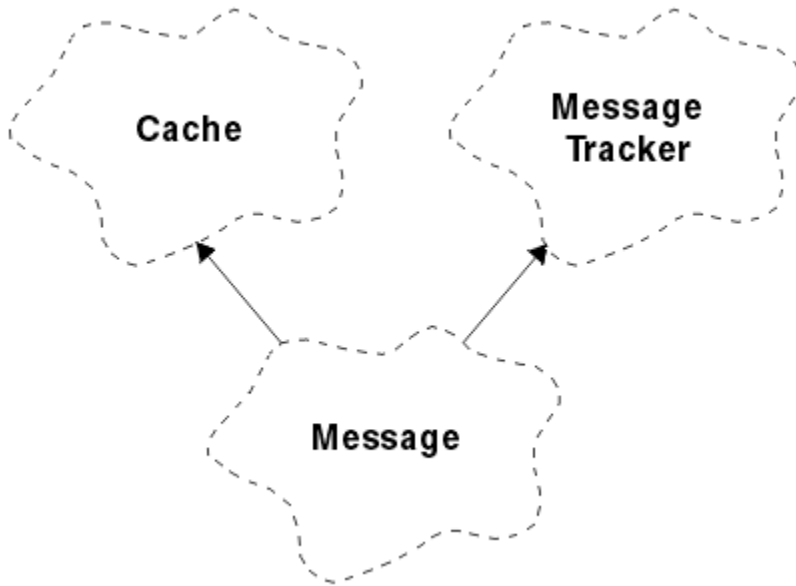


Figure 58. *ImqMessage* class

- [“Object attributes” on page 1870](#)
- [“Constructors” on page 1874](#)
- [“Object methods \(public\)” on page 1874](#)
- [“Object methods \(protected\)” on page 1876](#)
- [“Object data \(protected\)” on page 1876](#)

Object attributes

application id data

Identity information associated with a message. The initial value is a null string.

application origin data

Origin information associated with a message. The initial value is a null string.

backout count

The number of times that a message has been tentatively retrieved and subsequently backed out. The initial value is zero. This attribute is read-only.

character set

Coded Character Set Id. The initial value is MQCCSI_Q_MGR. The following additional values are possible:

- MQCCSI_INHERIT
- MQCCSI_EMBEDDED

You can also use a Coded Character Set Id of your choice. For information about this, see [“Code page conversion” on page 900](#).

encoding

The machine encoding of the message data. The initial value is MQENC_NATIVE.

expiry

A time-dependent quantity that controls how long IBM MQ retains an unretrieved message before discarding it. The initial value is MQEI_UNLIMITED.

format

The name of the format (template) that describes the layout of data in the buffer. Names longer than eight characters are truncated to eight characters. Names are always padded with blanks to eight characters. The initial constant value is MQFMT_NONE. The following additional constants are possible:

- MQFMT_ADMIN
- MQFMT_CICS
- MQFMT_COMMAND_1
- MQFMT_COMMAND_2
- MQFMT_DEAD_LETTER_HEADER
- MQFMT_DIST_HEADER
- MQFMT_EVENT
- MQFMT_IMS
- MQFMT_IMS_VAR_STRING
- MQFMT_MD_EXTENSION
- MQFMT_PCF
- MQFMT_REF_MSG_HEADER
- MQFMT_RF_HEADER
- MQFMT_STRING
- MQFMT_TRIGGER
- MQFMT_WORK_INFO_HEADER
- MQFMT_XMIT_Q_HEADER

You can also use an application-specific string of your choice. For more information about this, see the [“Format \(MQCHAR8\)”](#) on page 412 field of the message descriptor (MQMD).

message flags

Segmentation control information. The initial value is MQMF_SEGMENTATION_INHIBITED. The following additional values are possible:

- MQMF_SEGMENTATION_ALLOWED
- MQMF_MSG_IN_GROUP
- MQMF_LAST_MSG_IN_GROUP
- MQMF_SEGMENT
- MQMF_LAST_SEGMENT
- MQMF_NONE

message type

The broad categorization of a message. The initial value is MQMT_DATAGRAM. The following additional values are possible:

- MQMT_SYSTEM_FIRST
- MQMT_SYSTEM_LAST
- MQMT_DATAGRAM
- MQMT_REQUEST
- MQMT_REPLY
- MQMT_REPORT
- MQMT_APPL_FIRST
- MQMT_APPL_LAST

You can also use an application-specific value of your choice. For more information about this, see the [“MsgType \(MQLONG\)”](#) on page 423 field of the message descriptor (MQMD).

offset

Offset information. The initial value is zero.

original length

The original length of a segmented message. The initial value is MQOL_UNDEFINED.

persistence

Indicates that the message is important and must at all times be backed up using persistent storage. This option implies a performance penalty. The initial value is MQPER_PERSISTENCE_AS_Q_DEF. The following additional values are possible:

- MQPER_PERSISTENT
- MQPER_NOT_PERSISTENT

priority

The relative priority for transmission and delivery. Messages of the same priority are usually delivered in the same sequence as they were supplied (although there are several criteria that must be satisfied to guarantee this). The initial value is MQPRI_PRIORITY_AS_Q_DEF.

property validation

Specifies whether validation of properties should take place when a property of the message is set. The initial value is **MQCMHO_DEFAULT_VALIDATION**. The following additional values are possible:

- MQCMHO_VALIDATE
- MQCMHO_NO_VALIDATION

The following methods act on **property validation**:

MQLONG propertyValidation() const ;

Returns the **property validation** option.

void setPropertyValidation(const MQLONG option);

Sets the **property validation** option.

put application name

The name of the application that put a message. The initial value is a null string.

put application type

The type of application that put a message. The initial value is MQAT_NO_CONTEXT. The following additional values are possible:

- MQAT_AIX
- MQAT_CICS
- MQAT_CICS_BRIDGE
- MQAT_DOS
- MQAT_IMS
- MQAT_IMS_BRIDGE
- MQAT_MVS
- MQAT_NOTES_AGENT
- MQAT_OS2
- MQAT_OS390
- MQAT_OS400
- MQAT_QMGR
- MQAT_UNIX
- MQAT_WINDOWS
- MQAT_WINDOWS_NT

- MQAT_XCF
- MQAT_DEFAULT
- MQAT_UNKNOWN
- MQAT_USER_FIRST
- MQAT_USER_LAST

You can also use an application-specific string of your choice. For more information about this, see the [“PutApplType \(MQLONG\)”](#) on page 428 field of the message descriptor (MQMD).

put date

The date on which a message was put. The initial value is a null string.

put time

The time at which a message was put. The initial value is a null string.

reply-to queue manager name

The name of the queue manager to which any reply should be sent. The initial value is a null string.

reply-to queue name

The name of the queue to which any reply should be sent. The initial value is a null string.

report

Feedback information associated with a message. The initial value is MQRO_NONE. The following additional values are possible:

- MQRO_EXCEPTION
- MQRO_EXCEPTION_WITH_DATA
- MQRO_EXCEPTION_WITH_FULL_DATA *
- MQRO_EXPIRATION
- MQRO_EXPIRATION_WITH_DATA
- MQRO_EXPIRATION_WITH_FULL_DATA *
- MQRO_COA
- MQRO_COA_WITH_DATA
- MQRO_COA_WITH_FULL_DATA *
- MQRO_COD
- MQRO_COD_WITH_DATA
- MQRO_COD_WITH_FULL_DATA *
- MQRO_PAN
- MQRO_NAN
- MQRO_NEW_MSG_ID
- MQRO_NEW_CORREL_ID
- MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQRO_PASS_CORREL_ID
- MQRO_DEAD_LETTER_Q
- MQRO_DISCARD_MSG

where * indicates values that are not supported on IBM MQ for z/OS.

sequence number

Sequence information identifying a message within a group. The initial value is one.

total message length

The number of bytes that were available during the most recent attempt to read a message. This number will be greater than the `ImqCache message length` if the last message was truncated, or if the last message was not read because truncation would have occurred. This attribute is read-only. The initial value is zero.

This attribute can be useful in any situation involving truncated messages.

user id

A user identity associated with a message. The initial value is a null string.

Constructors

ImqMessage();

The default constructor.

ImqMessage(const ImqMessage & msg);

The copy constructor. See the **operator =** method for details.

Object methods (public)

void operator = (const ImqMessage & msg);

Copies the MQMD and message data from *msg*. If a buffer has been supplied by the user for this object, the amount of data copied is restricted to the available buffer size. Otherwise, the system ensures that a buffer of adequate size is made available for the copied data.

ImqString applicationIdData () const ;

Returns a copy of the **application id data**.

void setApplicationIdData (const char * data = 0);

Sets the **application id data**.

ImqString applicationOriginData () const ;

Returns a copy of the **application origin data**.

void setApplicationOriginData (const char * data = 0);

Sets the **application origin data**.

MQLONG backoutCount () const ;

Returns the **backout count**.

MQLONG characterSet () const ;

Returns the **character set**.

void setCharacterSet (const MQLONG ccsid = MQCCSI_Q_MGR);

Sets the **character set**.

MQLONG encoding () const ;

Returns the **encoding**.

void setEncoding (const MQLONG encoding = MQENC_NATIVE);

Sets the **encoding**.

MQLONG expiry () const ;

Returns the **expiry**.

void setExpiry (const MQLONG expiry);

Sets the **expiry**.

ImqString format () const ;

Returns a copy of the **format**, including trailing blanks.

ImqBoolean formatIs (const char * format-to-test) const ;

Returns TRUE if the **format** is the same as *format-to-test*.

void setFormat (const char * name = 0);

Sets the **format**, padded to eight characters with trailing blanks.

MQLONG messageFlags () const ;

Returns the **message flags**.

void setMessageFlags (const MQLONG flags);

Sets the **message flags**.

MQLONG messageType () const ;

Returns the **message type**.

void setMessageType (const MQLONG *type*);
 Sets the **message type**.

MQLONG offset () const ;
 Returns the **offset**.

void setOffset (const MQLONG *offset*);
 Sets the **offset**.

MQLONG originalLength () const ;
 Returns the **original length**.

void setOriginalLength (const MQLONG *length*);
 Sets the **original length**.

MQLONG persistence () const ;
 Returns the **persistence**.

void setPersistence (const MQLONG *persistence*);
 Sets the **persistence**.

MQLONG priority () const ;
 Returns the **priority**.

void setPriority (const MQLONG *priority*);
 Sets the **priority**.

ImqString putApplicationName () const ;
 Returns a copy of the **put application name**.

void setPutApplicationName (const char * *name* = 0);
 Sets the **put application name**.

MQLONG putApplicationType () const ;
 Returns the **put application type**.

void setPutApplicationType (const MQLONG *type* = MQAT_NO_CONTEXT);
 Sets the **put application type**.

ImqString putDate () const ;
 Returns a copy of the **put date**.

void setPutDate (const char * *date* = 0);
 Sets the **put date**.

ImqString putTime () const ;
 Returns a copy of the **put time**.

void setPutTime (const char * *time* = 0);
 Sets the **put time**.

ImqBoolean readItem (ImqItem & *item*);
 Reads into the *item* object from the message buffer, using the ImqItem **pasteIn** method. It returns TRUE if successful.

ImqString replyToQueueManagerName () const ;
 Returns a copy of the **reply-to queue manager name**.

void setReplyToQueueManagerName (const char * *name* = 0);
 Sets the **reply-to queue manager name**.

ImqString replyToQueueName () const ;
 Returns a copy of the **reply-to queue name**.

void setReplyToQueueName (const char * *name* = 0);
 Sets the **reply-to queue name**.

MQLONG report () const ;
 Returns the **report**.

void setReport (const MQLONG *report*);
 Sets the **report**.

MQLONG **sequenceNumber () const ;**

Returns the **sequence number**.

void **setSequenceNumber (const MQLONG *number*);**

Sets the **sequence number**.

size_t **totalMessageLength () const ;**

Returns the **total message length**.

ImqString **userId () const ;**

Returns a copy of the **user id**.

void **setUserId (const char * *id* = 0);**

Sets the **user id**.

ImqBoolean **writeItem (ImqItem & *item*);**

Writes from the *item* object into the message buffer, using the ImqItem **copyOut** method. Writing can take the form of insertion, replacement, or an append: this depends on the class of the *item* object.

This method returns TRUE if successful.

Object methods (protected)

static void **setVersionSupported (const MQLONG);**

Sets the **MQMD version**. Defaults to **MQMD_VERSION_2**.

Object data (protected)

MQMD1 *omqmd*

(IBM MQ for z/OS only.) The MQMD data structure.

MQMD2 *omqmd*

(Platforms other than z/OS.) The MQMD data structure.

ImqMessageTracker C++ class

This class encapsulates those attributes of an ImqMessage or ImqQueue object that can be associated with either object.

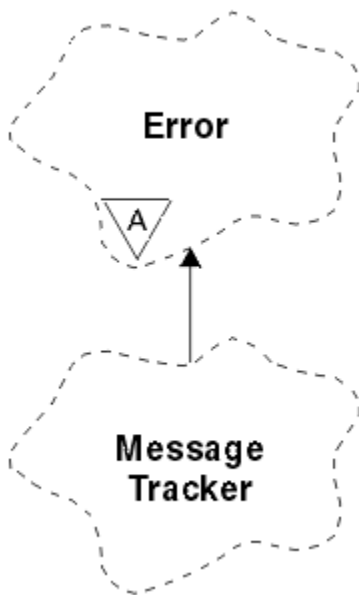


Figure 59. *ImqMessageTracker* class

This class relates to the MQI calls listed in [“ImqMessageTracker cross reference”](#) on page 1828.

- [“Object attributes”](#) on page 1877

- [“Constructors” on page 1878](#)
- [“Object methods \(public\)” on page 1878](#)
- [“Reason codes” on page 1879](#)

Object attributes

accounting token

A binary value (MQBYTE32) of length MQ_ACCOUNTING_TOKEN_LENGTH. The initial value is MQACT_NONE.

correlation id

A binary value (MQBYTE24) of length MQ_CORREL_ID_LENGTH that you assign to correlate messages. The initial value is MQCI_NONE. The additional value, MQCI_NEW_SESSION, is possible.

feedback

Feedback information to be sent with a message. The initial value is MQFB_NONE. The following additional values are possible:

- MQFB_SYSTEM_FIRST
- MQFB_SYSTEM_LAST
- MQFB_APPL_FIRST
- MQFB_APPL_LAST
- MQFB_COA
- MQFB_COD
- MQFB_EXPIRATION
- MQFB_PAN
- MQFB_NAN
- MQFB_QUIT
- MQFB_DATA_LENGTH_ZERO
- MQFB_DATA_LENGTH_NEGATIVE
- MQFB_DATA_LENGTH_TOO_BIG
- MQFB_BUFFER_OVERFLOW
- MQFB_LENGTH_OFF_BY_ONE
- MQFB_IIH_ERROR
- MQFB_NOT_AUTHORIZED_FOR_IMS
- MQFB_IMS_ERROR
- MQFB_IMS_FIRST
- MQFB_IMS_LAST
- MQFB_CICS_APPL_ABENDED
- MQFB_CICS_APPL_NOT_STARTED
- MQFB_CICS_BRIDGE_FAILURE
- MQFB_CICS_CCSID_ERROR
- MQFB_CICS_CIH_ERROR
- MQFB_CICS_COMMAREA_ERROR
- MQFB_CICS_CORREL_ID_ERROR
- MQFB_CICS_DLQ_ERROR
- MQFB_CICS_ENCODING_ERROR
- MQFB_CICS_INTERNAL_ERROR
- MQFB_CICS_NOT_AUTHORIZED

- MQFB_CICS_UOW_BACKED_OUT
- MQFB_CICS_UOW_ERROR

You can also use an application-specific string of your choice. For more information about this, see the [“Feedback \(MQLONG\)”](#) on page 409 field of the message descriptor (MQMD).

group id

A binary value (MQBYTE24) of length MQ_GROUP_ID_LENGTH unique within a queue. The initial value is MQGI_NONE.

message id

A binary value (MQBYTE24) of length MQ_MSG_ID_LENGTH unique within a queue. The initial value is MQMI_NONE.

Constructors

ImqMessageTracker();

The default constructor.

ImqMessageTracker(const ImqMessageTracker & tracker);

The copy constructor. See the **operator =** method for details.

Object methods (public)

void operator = (const ImqMessageTracker & tracker);

Copies instance data from *tracker*, replacing the existing instance data.

ImqBinary accountingToken () const ;

Returns a copy of the **accounting token**.

ImqBoolean setAccountingToken (const ImqBinary & token);

Sets the **accounting token**. The **data length** of *token* must be either zero or MQ_ACCOUNTING_TOKEN_LENGTH. This method returns TRUE if successful.

void setAccountingToken (const MQBYTE32 token = 0);

Sets the **accounting token**. *token* can be zero, which is the same as specifying MQACT_NONE. If *token* is nonzero, it must address MQ_ACCOUNTING_TOKEN_LENGTH bytes of binary data. When using predefined values such as MQACT_NONE, you might need to make a cast to ensure a signature match; for example, (MQBYTE *)MQACT_NONE.

ImqBinary correlationId () const ;

Returns a copy of the **correlation id**.

ImqBoolean setCorrelationId (const ImqBinary & token);

Sets the **correlation id**. The **data length** of *token* must be either zero or MQ_CORREL_ID_LENGTH. This method returns TRUE if successful.

void setCorrelationId (const MQBYTE24 id = 0);

Sets the **correlation id**. *id* can be zero, which is the same as specifying MQCI_NONE. If *id* is nonzero, it must address MQ_CORREL_ID_LENGTH bytes of binary data. When using predefined values such as MQCI_NONE, you might need to make a cast to ensure a signature match; for example, (MQBYTE *)MQCI_NONE.

MQLONG feedback () const ;

Returns the **feedback**.

void setFeedback (const MQLONG feedback);

Sets the **feedback**.

ImqBinary groupId () const ;

Returns a copy of the **group id**.

ImqBoolean setGroupId (const ImqBinary & token);

Sets the **group id**. The **data length** of *token* must be either zero or MQ_GROUP_ID_LENGTH. This method returns TRUE if successful.

void setGroupId (const MQBYTE24 id = 0);

Sets the **group id**. *id* can be zero, which is the same as specifying MQGI_NONE. If *id* is nonzero, it must address MQ_GROUP_ID_LENGTH bytes of binary data. When using predefined values such as MQGI_NONE, you might need to make a cast to ensure a signature match, for example (MQBYTE *)MQGI_NONE.

ImqBinary messageId () const ;

Returns a copy of the **message id**.

ImqBoolean setMessageId (const ImqBinary & token);

Sets the **message id**. The **data length** of *token* must be either zero or MQ_MSG_ID_LENGTH. This method returns TRUE if successful.

void setMessageId (const MQBYTE24 id = 0);

Sets the **message id**. *id* can be zero, which is the same as specifying MQMI_NONE. If *id* is nonzero, it must address MQ_MSG_ID_LENGTH bytes of binary data. When using predefined values such as MQMI_NONE, you might need to make a cast to ensure a signature match, for example (MQBYTE *)MQMI_NONE.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR

ImqNamelist C++ class

This class encapsulates a namelist.

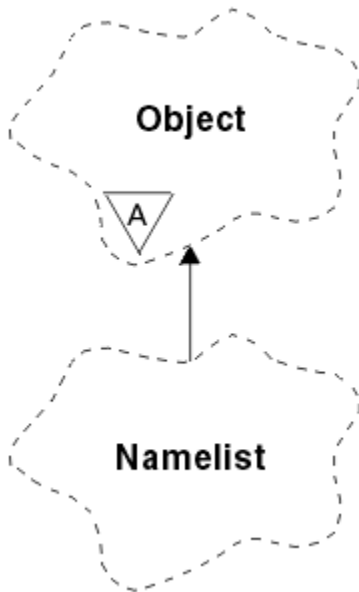


Figure 60. ImqNamelist class

This class relates to the MQI calls listed in [“ImqNamelist cross reference” on page 1828](#).

- [“Object attributes” on page 1879](#)
- [“Constructors” on page 1880](#)
- [“Object methods \(public\)” on page 1880](#)
- [“Reason codes” on page 1880](#)

Object attributes

name count

The number of object names in **namelist names**. This attribute is read-only.

namelist names

Object names, the number of which is indicated by the **name count**. This attribute is read-only.

Constructors

ImqNamelist();

The default constructor.

ImqNamelist(const ImqNamelist & list);

The copy constructor. The ImqObject **open status** is false.

ImqNamelist(const char * name);

Sets the ImqObject name to **name**.

Object methods (public)

void operator = (const ImqNamelist & list);

Copies instance data from *list*, replacing the existing instance data. The ImqObject **open status** is false.

ImqBoolean nameCount(MQLONG & count);

Provides a copy of the **name count**. It returns TRUE if successful.

MQLONG nameCount ();

Returns the **name count** without any indication of possible errors.

ImqBoolean namelistName (const MQLONG index, ImqString & name);

Provides a copy of one the **namelist names** by zero based index. It returns TRUE if successful.

ImqString namelistName (const MQLONG index);

Returns one of the **namelist names** by zero-based index without any indication of possible errors.

Reason codes

- MQRC_INDEX_ERROR
- MQRC_INDEX_NOT_PRESENT

ImqObject C++ class

This class is abstract. When an object of this class is destroyed, it is automatically closed, and its ImqQueueManager connection severed.

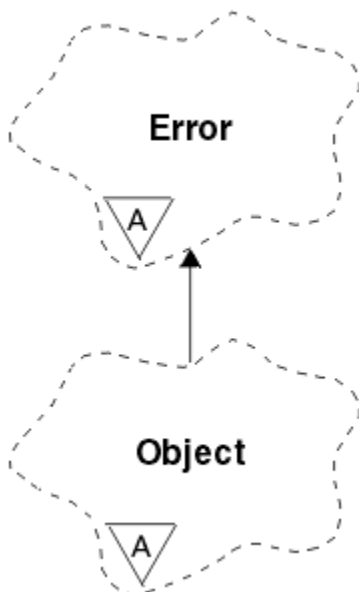


Figure 61. ImqObject class

This class relates to the MQI calls listed in [“ImqObject cross reference” on page 1828](#).

- [“Class attributes” on page 1881](#)
- [“Object attributes” on page 1881](#)
- [“Constructors” on page 1882](#)
- [“Class methods \(public\)” on page 1882](#)
- [“Object methods \(public\)” on page 1882](#)
- [“Object methods \(protected\)” on page 1884](#)
- [“Object data \(protected\)” on page 1885](#)
- [“Reason codes” on page 1885](#)
-

Class attributes

behavior

Controls the behavior of implicit opening.

IMQ_IMPL_OPEN (8L)

Implicit opening is allowed. This is the default.

Object attributes

alteration date

The alteration date. This attribute is read-only.

alteration time

The alteration time. This attribute is read-only.

alternate user id

The alternate user id, up to MQ_USER_ID_LENGTH characters. The initial value is a null string.

alternate security id

The alternate security id. A binary value (MQBYTE40) of length MQ_SECURITY_ID_LENGTH. The initial value is MQSID_NONE.

close options

Options that apply when an object is closed. The initial value is MQCO_NONE. This attribute is ignored during implicit reopen operations, where a value of MQCO_NONE is always used.

connection reference

A reference to an ImqQueueManager object that provides the required connection to a (local) queue manager. For an ImqQueueManager object, it is the object itself. The initial value is zero.

Note: Do not confuse this with the **queue manager name** that identifies a queue manager (possibly remote) for a named queue.

description

The descriptive name (up to 64 characters) of the queue manager, queue, namelist, or process. This attribute is read-only.

name

The name (up to 48 characters) of the queue manager, queue, namelist, or process. The initial value is a null string. The name of a model queue changes after an **open** to the name of the resulting dynamic queue.

Note: An ImqQueueManager can have a null name, representing the default queue manager. The name changes to the actual queue manager after a successful **open**. An ImqDistributionList is dynamic and must have a null name.

next managed object

This is the next object of this class, in no particular order, having the same **connection reference** as this object. The initial value is zero.

open options

Options that apply when an object is opened. The initial value is MQOO_INQUIRE. There are two ways to set appropriate values:

1. Do not set the **open options** and do not use the **open** method. IBM MQ automatically adjusts the **open options** and automatically opens, reopens, and closes objects as required. This can result in unnecessary reopen operations, because IBM MQ uses the **openFor** method, and this adds **open options** incrementally only.
2. Set the **open options** before using any methods that result in an MQI call (see [“C++ and MQI cross-reference” on page 1822](#)). This ensures that unnecessary reopen operations do not occur. Set open options explicitly if any of the potential reopen problems are likely to occur (see [Reopen](#)).

If you use the **open** method, you *must* ensure that the **open options** are appropriate first. However, using the **open** method is not mandatory; IBM MQ still exhibits the same behavior as in case 1, but in this circumstance, the behavior is efficient.

Zero is not a valid value; set the appropriate value before attempting to open the object. This can be done using either **setOpenOptions** (*IOpenOptions*) followed by **open** (), or **openFor** (*IRequiredOpenOption*).

Note:

1. MQOO_OUTPUT is substituted for MQOO_INQUIRE during the **open** method for a distribution list, as MQOO_OUTPUT is the only valid **open option** at this time. However, it is good practice always to set MQOO_OUTPUT explicitly in application programs that use the **open** method.
2. Specify MQOO_RESOLVE_NAMES if you want to use the **resolved queue manager name** and **resolved queue name** attributes of the class.

open status

Whether the object is open (TRUE) or closed (FALSE). The initial value is FALSE. This attribute is read-only.

previous managed object

The previous object of this class, in no particular order, having the same **connection reference** as this object. The initial value is zero.

queue manager identifier

The queue manager identifier. This attribute is read-only.

Constructors

ImqObject();

The default constructor.

ImqObject(const ImqObject & *object*);

The copy constructor. The **open status** will be FALSE.

Class methods (public)

static MQLONG behavior();

Returns the **behavior**.

void setBehavior(const MQLONG *behavior* = 0);

Sets the **behavior**.

Object methods (public)

void operator = (const ImqObject & *object*);

Performs a close if necessary, and copies the instance data from *object*. The **open status** will be FALSE.

ImqBoolean alterationDate(ImqString & *date*);

Provides a copy of the **alteration date**. It returns TRUE if successful.

ImqString alterationDate();

Returns the **alteration date** without any indication of possible errors.

ImqBoolean alterationTime(ImqString & time);

Provides a copy of the **alteration time**. It returns TRUE if successful.

ImqString alterationTime();

Returns the **alteration time** without any indication of possible errors.

ImqString alternateUserId () const ;

Returns a copy of the **alternate user id**.

ImqBoolean setAlternateUserId (const char * id);

Sets the **alternate user id**. The **alternate user id** can be set only while the **open status** is FALSE. This method returns TRUE if successful.

ImqBinary alternateSecurityId () const ;

Returns a copy of the **alternate security id**.

ImqBoolean setAlternateSecurityId(const ImqBinary & token);

Sets the **alternate security id**. The **alternate security id** can be set only while the **open status** is FALSE. The data length of *token* must be either zero or MQ_SECURITY_ID_LENGTH. It returns TRUE if successful.

ImqBoolean setAlternateSecurityId(const MQBYTE* token = 0);

Sets the **alternate security id**. *token* can be zero, which is the same as specifying MQSID_NONE.

If *token* is nonzero, it must address MQ_SECURITY_ID_LENGTH bytes of binary data. When using predefined values such as MQSID_NONE, you might need to make a cast to ensure signature match; for example, (MQBYTE *)MQSID_NONE.

The **alternate security id** can be set only while the **open status** is TRUE. It returns TRUE if successful.

ImqBoolean setAlternateSecurityId(const unsigned char * id = 0);

Sets the **alternate security id**.

ImqBoolean close ();

Sets the **open status** to FALSE. It returns TRUE if successful.

MQLONG closeOptions () const ;

Returns the **close options**.

void setCloseOptions (const MQLONG options);

Sets the **close options**.

ImqQueueManager * connectionReference () const ;

Returns the **connection reference**.

void setConnectionReference (ImqQueueManager & manager);

Sets the **connection reference**.

void setConnectionReference (ImqQueueManager * manager = 0);

Sets the **connection reference**.

virtual ImqBoolean description (ImqString & description) = 0 ;

Provides a copy of the **description**. It returns TRUE if successful.

ImqString description ();

Returns a copy of the **description** without any indication of possible errors.

virtual ImqBoolean name (ImqString & name);

Provides a copy of the **name**. It returns TRUE if successful.

ImqString name ();

Returns a copy of the **name** without any indication of possible errors.

ImqBoolean setName (const char * name = 0);

Sets the **name**. The **name** can only be set while the **open status** is FALSE, and, for an ImqQueueManager, while the **connection status** is FALSE. It returns TRUE if successful.

ImqObject * nextManagedObject () const ;

Returns the **next managed object**.

ImqBoolean open ();

Changes the **open status** to TRUE by opening the object as necessary, using among other attributes the **open options** and the **name**. This method uses the **connection reference** information and the ImqQueueManager **connect** method if necessary to ensure that the ImqQueueManager **connection status** is TRUE. It returns the **open status**.

ImqBoolean openFor (const MQLONG *required-options* = 0);

Attempts to ensure that the object is open with **open options**, or with **open options** that guarantee the behavior implied by the *required-options* parameter value.

If *required-options* is zero, input is required, and any input option suffices. So, if the **open options** already contain one of:

- MQOO_INPUT_AS_Q_DEF
- MQOO_INPUT_SHARED
- MQOO_INPUT_EXCLUSIVE

the **open options** are already satisfactory and are not changed; if the **open options** do not already contain any of these options, MQOO_INPUT_AS_Q_DEF is set in the **open options**.

If *required-options* is nonzero, the required options are added to the **open options** ; if *required-options* is any of these options, the others are reset.

If any of the **open options** are changed and the object is already open, the object is closed temporarily and reopened in order to adjust the **open options**.

It returns TRUE if successful. Success indicates that the object is open with appropriate options.

MQLONG openOptions () const ;

Returns the **open options**.

ImqBoolean setOpenOptions (const MQLONG *options*);

Sets the **open options**. The **open options** can be set only while the **open status** is FALSE. It returns TRUE if successful.

ImqBoolean openStatus () const ;

Returns the **open status**.

ImqObject * previousManagedObject () const ;

Returns the **previous managed object**.

ImqBoolean queueManagerIdentifier(ImqString & *id*);

Provides a copy of the **queue manager identifier**. It returns TRUE if successful.

ImqString queueManagerIdentifier();

Returns the **queue manager identifier** without any indication of possible errors.

Object methods (protected)**virtual ImqBoolean closeTemporarily ();**

Closes an object safely before reopening. It returns TRUE if successful. This method assumes that the **open status** is TRUE.

MQHCONN connectionHandle () const ;

Returns the MQHCONN associated with the **connection reference**. This value is zero if there is no **connection reference** or if the Manager is not connected.

ImqBoolean inquire (const MQLONG *int-attr*, MQLONG & *value*);

Returns an integer value, the index of which is an MQIA_* value. In case of error, the value is set to MQIAV_UNDEFINED.

ImqBoolean inquire (const MQLONG *char-attr*, char * & *buffer*, const size_t *length*);

Returns a character string, the index of which is an MQCA_* value.

Note: Both of these methods return only a single attribute value. If a *snapshot* is required of more than one value, where the values are consistent with each other for an instant, IBM MQ C++ does not provide this facility and you must use the MQINQ call with appropriate parameters.

virtual void openInformationDisperse ();

Disperses information from the variable section of the MQOD data structure immediately after an MQOPEN call.

virtual ImqBoolean openInformationPrepare ();

Prepares information for the variable section of the MQOD data structure immediately before an MQOPEN call, and returns TRUE if successful.

ImqBoolean set (const MQLONG int-attr, const MQLONG value);

Sets an IBM MQ integer attribute.

ImqBoolean set (const MQLONG char-attr, const char * buffer, const size_t required-length);

Sets an IBM MQ character attribute.

void setNextManagedObject (const ImqObject * object = 0);

Sets the **next managed object**.

Attention: Use this function only if you are sure it will not break the managed object list.

void setPreviousManagedObject (const ImqObject * object = 0);

Sets the **previous managed object**.

Attention: Use this function only if you are sure it will not break the managed object list.

Object data (protected)**MQHOBJ ohobj**

The IBM MQ object handle (valid only when **open status** is TRUE).

MQOD omqod

The embedded MQOD data structure. The amount of storage allocated for this data structure is that required for an MQOD Version 2. Inspect the version number (*omqod.Version*) and access the other fields as follows:

MQOD_VERSION_1

All other fields in *omqod* can be accessed.

MQOD_VERSION_2

All other fields in *omqod* can be accessed.

MQOD_VERSION_3

omqod.pmqod is a pointer to a dynamically allocated, larger, MQOD. No other fields in *omqod* can be accessed. All fields addressed by *omqod.pmqod* can be accessed.

Note: *omqod.pmqod.Version* can be less than *omqod.Version*, indicating that the IBM MQ MQI client has more functionality than the IBM MQ server.

Reason codes

- MQRC_ATTRIBUTE_LOCKED
- MQRC_INCONSISTENT_OBJECT_STATE
- MQRC_NO_CONNECTION_REFERENCE
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_REOPEN_SAVED_CONTEXT_ERR
- (reason codes from MQCLOSE)
- (reason codes from MQCONN)
- (reason codes from MQINQ)
- (reason codes from MQOPEN)
- (reason codes from MQSET)

ImqProcess C++ class

This class encapsulates an application process (an IBM MQ object of type MQOT_PROCESS) that can be triggered by a trigger monitor.

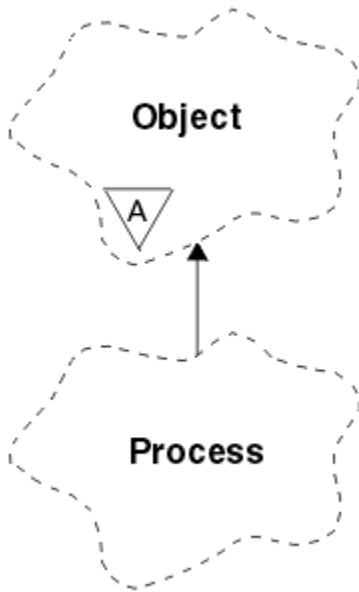


Figure 62. ImqProcess class

- [“Object attributes” on page 1886](#)
- [“Constructors” on page 1886](#)
- [“Object methods \(public\)” on page 1886](#)

Object attributes

application id

The identity of the application process. This attribute is read-only.

application type

The type of the application process. This attribute is read-only.

environment data

The environment information for the process. This attribute is read-only.

user data

User data for the process. This attribute is read-only.

Constructors

ImqProcess();

The default constructor.

ImqProcess(const ImqProcess & process);

The copy constructor. The ImqObject **open status** is FALSE.

ImqProcess(const char * name);

Sets the ImqObject **name**.

Object methods (public)

void operator = (const ImqProcess & process);

Performs a close if necessary, and then copies instance data from *process*. The ImqObject **open status** will be FALSE.

ImqBoolean applicationId (ImqString & id);

Provides a copy of the **application id**. It returns TRUE if successful.

ImqString applicationId ();

Returns the **application id** without any indication of possible errors.

ImqBoolean applicationType (MQLONG & type);

Provides a copy of the **application type**. It returns TRUE if successful.

MQLONG applicationType ();

Returns the **application type** without any indication of possible errors.

ImqBoolean environmentData (ImqString & data);

Provides a copy of the **environment data**. It returns TRUE if successful.

ImqString environmentData ();

Returns the **environment data** without any indication of possible errors.

ImqBoolean userData (ImqString & data);

Provides a copy of the **user data**. It returns TRUE if successful.

ImqString userData ();

Returns the **user data** without any indication of possible errors.

ImqPutMessageOptions C++ class

This class encapsulates the MQPMO data structure.

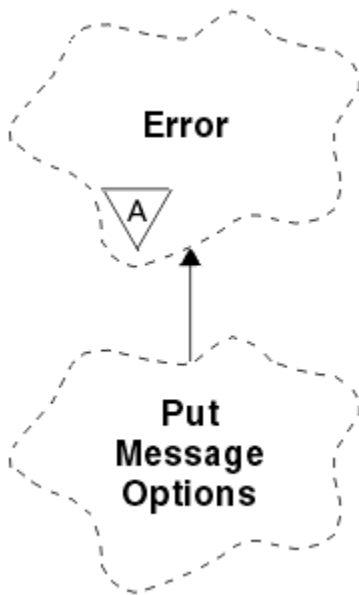


Figure 63. *ImqPutMessageOptions* class

- [“Object attributes” on page 1887](#)
- [“Constructors” on page 1888](#)
- [“Object methods \(public\)” on page 1888](#)
- [“Object data \(protected\)” on page 1889](#)
- [“Reason codes” on page 1889](#)

Object attributes

context reference

An ImqQueue that provides a context for messages. Initially there is no reference.

options

The put message options. The initial value is MQPMO_NONE. The following additional values are possible:

- MQPMO_SYNCPOINT
- MQPMO_NO_SYNCPOINT
- MQPMO_NEW_MSG_ID
- MQPMO_NEW_CORREL_ID
- MQPMO_LOGICAL_ORDER
- MQPMO_NO_CONTEXT
- MQPMO_DEFAULT_CONTEXT
- MQPMO_PASS_IDENTITY_CONTEXT
- MQPMO_PASS_ALL_CONTEXT
- MQPMO_SET_IDENTITY_CONTEXT
- MQPMO_SET_ALL_CONTEXT
- MQPMO_ALTERNATE_USER_AUTHORITY
- MQPMO_FAIL_IF QUIESCING

record fields

The flags that control the inclusion of put message records when a message is put. The initial value is MQPMRF_NONE. The following additional values are possible:

- MQPMRF_MSG_ID
- MQPMRF_CORREL_ID
- MQPMRF_GROUP_ID
- MQPMRF_FEEDBACK
- MQPMRF_ACCOUNTING_TOKEN

ImqMessageTracker attributes are taken from the object for any field that is specified.

ImqMessageTracker attributes are taken from the ImqMessage object for any field that is *not* specified.

resolved queue manager name

Name of a destination queue manager determined during a put. The initial value is null. This attribute is read-only.

resolved queue name

Name of a destination queue determined during a put. The initial value is null. This attribute is read-only.

syncpoint participation

TRUE when messages are put under syncpoint control.

Constructors

ImqPutMessageOptions();

The default constructor.

ImqPutMessageOptions(const ImqPutMessageOptions & pmo);

The copy constructor.

Object methods (public)

void operator = (const ImqPutMessageOptions & pmo);

Copies instance data from *pmo*, replacing the existing instance data.

ImqQueue * contextReference () const ;

Returns the **context reference**.

void setContextReference (const ImqQueue & *queue*);

Sets the **context reference**.

void setContextReference (const ImqQueue * *queue* = 0);

Sets the **context reference**.

MQLONG options () const ;

Returns the **options**.

void setOptions (const MQLONG *options*);

Sets the **options**, including the **syncpoint participation** value.

MQLONG recordFields () const ;

Returns the **record fields**.

void setRecordFields (const MQLONG *fields*);

Sets the **record fields**.

ImqString resolvedQueueManagerName () const ;

Returns a copy of the **resolved queue manager name**.

ImqString resolvedQueueName () const ;

Returns a copy of the **resolved queue name**.

ImqBoolean syncPointParticipation () const ;

Returns the **syncpoint participation** value, which is TRUE if the **options** include MQPMO_SYNCPOINT.

void setSyncPointParticipation (const ImqBoolean *sync*);

Sets the **syncpoint participation** value. If *sync* is TRUE, the **options** are altered to include MQPMO_SYNCPOINT, and to exclude MQPMO_NO_SYNCPOINT. If *sync* is FALSE, the **options** are altered to include MQPMO_NO_SYNCPOINT, and to exclude MQPMO_SYNCPOINT.

Object data (protected)

MQPMO *omqpmo*

The MQPMO data structure.

Reason codes

- MQRC_STORAGE_NOT_AVAILABLE

ImqQueue C++ class

This class encapsulates a message queue (an IBM MQ object of type MQOT_Q).

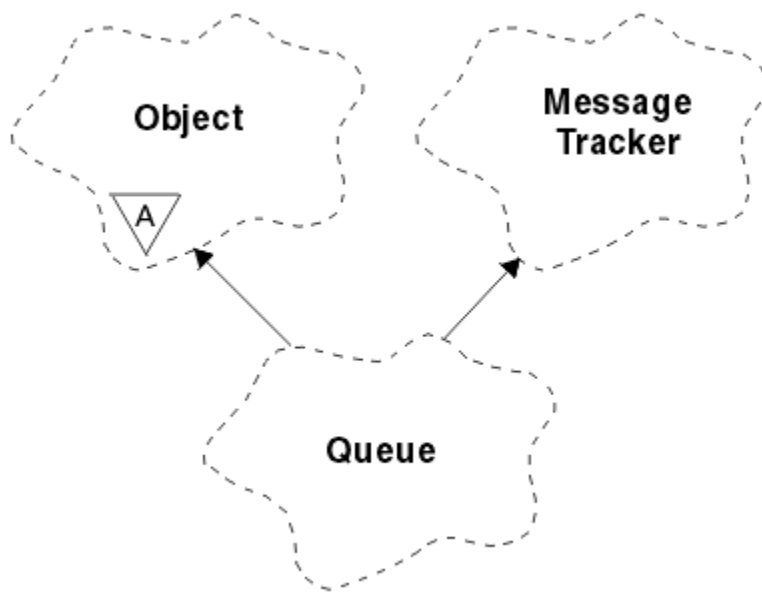


Figure 64. *ImqQueue* class

This class relates to the MQI calls listed in [Table 674 on page 1829](#).

- [“Object attributes” on page 1890](#)
- [“Constructors” on page 1893](#)
- [“Object methods \(public\)” on page 1893](#)
- [“Object methods \(protected\)” on page 1899](#)
- [“Reason codes” on page 1899](#)

Object attributes

backout requeue name

Excessive backout requeue name. This attribute is read-only.

backout threshold

Backout threshold. This attribute is read-only.

base queue name

Name of the queue that the alias resolves to. This attribute is read-only.

cluster name

Cluster name. This attribute is read-only.

cluster namelist name

Cluster namelist name. This attribute is read-only.

cluster workload rank

Cluster workload rank. This attribute is read-only.

cluster workload priority

Cluster workload priority. This attribute is read-only.

cluster workload use queue

Cluster workload use queue value. This attribute is read-only.

creation date

Queue creation data. This attribute is read-only.

creation time

Queue creation time. This attribute is read-only.

current depth

Number of messages on the queue. This attribute is read-only.

default bind

Default bind. This attribute is read-only.

default input open option

Default open-for-input option. This attribute is read-only.

default persistence

Default message persistence. This attribute is read-only.

default priority

Default message priority. This attribute is read-only.

definition type

Queue definition type. This attribute is read-only.

depth high event

Control attribute for queue depth high events. This attribute is read-only.

depth high limit

High limit for the queue depth. This attribute is read-only.

depth low event

Control attribute for queue depth low events. This attribute is read-only.

depth low limit

Low limit for the queue depth. This attribute is read-only.

depth maximum event

Control attribute for queue depth maximum events. This attribute is read-only.

distribution list reference

Optional reference to an `ImqDistributionList` that can be used to distribute messages to more than one queue, including this one. The initial value is null.

Note: When an `ImqQueue` object is opened, any open `ImqDistributionList` object that it references is automatically closed.

distribution lists

The capability of a transmission queue to support distribution lists. This attribute is read-only.

dynamic queue name

Dynamic queue name. The initial value is `AMQ.*` for all Windows, UNIX, and Linux platforms.

harden get backout

Whether to harden the backout count. This attribute is read-only.

index type

Index type. This attribute is read-only.

inhibit get

Whether get operations are allowed. The initial value is dependent on the queue definition. This attribute is valid for an alias or local queue only.

inhibit put

Whether put operations are allowed. The initial value is dependent on the queue definition.

initiation queue name

Name of the initiation queue. This attribute is read-only.

maximum depth

Maximum number of messages allowed on the queue. This attribute is read-only.

maximum message length

Maximum length for any message on this queue, which can be less than the maximum for any queue managed by the associated queue manager. This attribute is read-only.

message delivery sequence

Whether message priority is relevant. This attribute is read-only.

next distributed queue

Next object of this class, in no particular order, having the same **distribution list reference** as this object. The initial value is zero.

If an object in a chain is deleted, the previous object and next object are updated so that their distributed queue links no longer point to the deleted object.

non-persistent message class

Level of reliability for non-persistent messages put to this queue. This attribute is read-only.

open input count

Number of ImqQueue objects that are open for input. This attribute is read-only.

open output count

Number of ImqQueue objects that are open for output. This attribute is read-only.

previous distributed queue

Previous object of this class, in no particular order, having the same **distribution list reference** as this object. The initial value is zero.

If an object in a chain is deleted, the previous object and next object are updated so that their distributed queue links no longer point to the deleted object.

process name

Name of the process definition. This attribute is read-only.

queue accounting

Level of accounting information for queues. This attribute is read-only.

queue manager name

Name of the queue manager (possibly remote) where the queue resides. Do not confuse the queue manager named here with the ImqObject **connection reference**, which references the (local) queue manager providing a connection. The initial value is null.

queue monitoring

Level of monitoring data collection for the queue. This attribute is read-only.

queue statistics

Level of statistics data for the queue. This attribute is read-only.

queue type

Queue type. This attribute is read-only.

remote queue manager name

Name of the remote queue manager. This attribute is read-only.

remote queue name

Name of the remote queue as known on the remote queue manager. This attribute is read-only.

resolved queue manager name

Resolved queue manager name. This attribute is read-only.

resolved queue name

Resolved queue name. This attribute is read-only.

retention interval

Queue retention interval. This attribute is read-only.

scope

Scope of the queue definition. This attribute is read-only.

service interval

Service interval. This attribute is read-only.

service interval event

Control attribute for service interval events. This attribute is read-only.

shareability

Whether the queue can be shared. This attribute is read-only.

storage class

Storage class. This attribute is read-only.

transmission queue name

Name of the transmission queue. This attribute is read-only.

trigger control

Trigger control. The initial value depends on the queue definition. This attribute is valid for a local queue only.

trigger data

Trigger data. The initial value depends on the queue definition. This attribute is valid for a local queue only.

trigger depth

Trigger depth. The initial value depends on the queue definition. This attribute is valid for a local queue only.

trigger message priority

Threshold message priority for triggers. The initial value depends on the queue definition. This attribute is valid for a local queue only.

trigger type

Trigger type. The initial value depends on the queue definition. This attribute is valid for a local queue only.

usage

Usage. This attribute is read-only.

Constructors**ImqQueue();**

The default constructor.

ImqQueue(const ImqQueue & *queue*);

The copy constructor. The ImqObject **open status** will be FALSE.

ImqQueue(const char * *name*);

Sets the ImqObject **name**.

Object methods (public)**void operator = (const ImqQueue & *queue*);**

Performs a close if necessary, and then copies instance data from *queue*. The ImqObject **open status** will be FALSE.

ImqBoolean backoutRequeueName (ImqString & *name*);

Provides a copy of the **backout requeue name**. It returns TRUE if successful.

ImqString backoutRequeueName ();

Returns the **backout requeue name** without any indication of possible errors.

ImqBoolean backoutThreshold (MQLONG & *threshold*);

Provides a copy of the **backout threshold**. It returns TRUE if successful.

MQLONG backoutThreshold ();

Returns the **backout threshold** value without any indication of possible errors.

ImqBoolean baseQueueName (ImqString & *name*);

Provides a copy of the **base queue name**. It returns TRUE if successful.

ImqString baseQueueName ();

Returns the **base queue name** without any indication of possible errors.

ImqBoolean clusterName(ImqString & *name*);

Provides a copy of the **cluster name**. It returns TRUE if successful.

ImqString clusterName();

Returns the **cluster name** without any indication of possible errors.

ImqBoolean clusterNamelistName(ImqString & name);

Provides a copy of the **cluster namelist name**. It returns TRUE if successful.

ImqString clusterNamelistName();

Returns the **cluster namelist name** without any indication of errors.

ImqBoolean clusterWorkLoadPriority (MQLONG & priority);

Provides a copy of the cluster workload priority value. It returns TRUE if successful.

MQLONG clusterWorkLoadPriority ();

Returns the cluster workload priority value without any indication of possible errors.

ImqBoolean clusterWorkLoadRank (MQLONG & rank);

Provides a copy of the cluster workload rank value. It returns TRUE if successful.

MQLONG clusterWorkLoadRank ();

Returns the cluster workload rank value without any indication of possible errors.

ImqBoolean clusterWorkLoadUseQ (MQLONG & useq);

Provides a copy of the cluster workload use queue value. It returns TRUE if successful.

MQLONG clusterWorkLoadUseQ ();

Returns the cluster workload use queue value without any indication of possible errors.

ImqBoolean creationDate (ImqString & date);

Provides a copy of the **creation date**. It returns TRUE if successful.

ImqString creationDate ();

Returns the **creation date** without any indication of possible errors.

ImqBoolean creationTime (ImqString & time);

Provides a copy of the **creation time**. It returns TRUE if successful.

ImqString creationTime ();

Returns the **creation time** without any indication of possible errors.

ImqBoolean currentDepth (MQLONG & depth);

Provides a copy of the **current depth**. It returns TRUE if successful.

MQLONG currentDepth ();

Returns the **current depth** without any indication of possible errors.

ImqBoolean defaultInputOpenOption (MQLONG & option);

Provides a copy of the **default input open option**. It returns TRUE if successful.

MQLONG defaultInputOpenOption ();

Returns the **default input open option** without any indication of possible errors.

ImqBoolean defaultPersistence (MQLONG & persistence);

Provides a copy of the **default persistence**. It returns TRUE if successful.

MQLONG defaultPersistence ();

Returns the **default persistence** without any indication of possible errors.

ImqBoolean defaultPriority (MQLONG & priority);

Provides a copy of the **default priority**. It returns TRUE if successful.

MQLONG defaultPriority ();

Returns the **default priority** without any indication of possible errors.

ImqBoolean defaultBind (MQLONG & bind);

Provides a copy of the **default bind**. It returns TRUE if successful.

MQLONG defaultBind ();

Returns the **default bind** without any indication of possible errors.

ImqBoolean definitionType (MQLONG & type);

Provides a copy of the **definition type**. It returns TRUE if successful.

MQLONG definitionType ();

Returns the **definition type** without any indication of possible errors.

ImqBoolean depthHighEvent (MQLONG & event);

Provides a copy of the enablement state of the **depth high event**. It returns TRUE if successful.

MQLONG depthHighEvent ();

Returns the enablement state of the **depth high event** without any indication of possible errors.

ImqBoolean depthHighLimit (MQLONG & limit);

Provides a copy of the **depth high limit**. It returns TRUE if successful.

MQLONG depthHighLimit ();

Returns the **depth high limit** value without any indication of possible errors.

ImqBoolean depthLowEvent (MQLONG & event);

Provides a copy of the enablement state of the **depth low event**. It returns TRUE if successful.

MQLONG depthLowEvent ();

Returns the enablement state of the **depth low event** without any indication of possible errors.

ImqBoolean depthLowLimit (MQLONG & limit);

Provides a copy of the **depth low limit**. It returns TRUE if successful.

MQLONG depthLowLimit ();

Returns the **depth low limit** value without any indication of possible errors.

ImqBoolean depthMaximumEvent (MQLONG & event);

Provides a copy of the enablement state of the **depth maximum event**. It returns TRUE if successful.

MQLONG depthMaximumEvent ();

Returns the enablement state of the **depth maximum event** without any indication of possible errors.

ImqDistributionList * distributionListReference () const ;

Returns the **distribution list reference**.

void setDistributionListReference (ImqDistributionList & list);

Sets the **distribution list reference**.

void setDistributionListReference (ImqDistributionList * list = 0);

Sets the **distribution list reference**.

ImqBoolean distributionLists (MQLONG & support);

Provides a copy of the **distribution lists** value. It returns TRUE if successful.

MQLONG distributionLists ();

Returns the **distribution lists** value without any indication of possible errors.

ImqBoolean setDistributionLists (const MQLONG support);

Sets the **distribution lists** value. It returns TRUE if successful.

ImqString dynamicQueueName () const ;

Returns a copy of the **dynamic queue name**.

ImqBoolean setDynamicQueueName (const char * name);

Sets the **dynamic queue name**. The **dynamic queue name** can be set only while the ImqObject **open status** is FALSE. It returns TRUE if successful.

ImqBoolean get (ImqMessage & msg, ImqGetMessageOptions & options);

Retrieves a message from the queue, using the specified *options*. Invokes the ImqObject **openFor** method if necessary to ensure that the ImqObject **open options** include either one of the MQOO_INPUT_* values, or the MQOO_BROWSE value, depending on the *options*. If the *msg* object has an ImqCache **automatic buffer**, the buffer grows to accommodate any message retrieved. The **clearMessage** method is invoked against the *msg* object before retrieval.

This method returns TRUE if successful.

Note: The result of the method invocation is FALSE if the ImqObject **reason code** is MQRC_TRUNCATED_MSG_FAILED, even though this **reason code** is classified as a warning. If a truncated message is accepted, the ImqCache **message length** reflects the truncated length. In either event, the ImqMessage **total message length** indicates the number of bytes that were available.

ImqBoolean get (ImqMessage & msg);

As for the previous method, except that default get message options are used.

ImqBoolean get (ImqMessage & msg, ImqGetMessageOptions & options, const size_t buffer-size);
 As for the previous two methods, except that an overriding *buffer-size* is indicated. If the *msg* object employs an ImqCache **automatic buffer**, the **resizeBuffer** method is invoked on the *msg* object prior to message retrieval, and the buffer does not grow further to accommodate any larger message.

ImqBoolean get (ImqMessage & msg, const size_t buffer-size);
 As for the previous method, except that default get message options are used.

ImqBoolean hardenGetBackout (MQLONG & harden);
 Provides a copy of the **harden get backout** value. It returns TRUE if successful.

MQLONG hardenGetBackout ();
 Returns the **harden get backout** value without any indication of possible errors.

ImqBoolean indexType(MQLONG & type);
 Provides a copy of the **index type**. It returns TRUE if successful.

MQLONG indexType();
 Returns the **index type** without any indication of possible errors.

ImqBoolean inhibitGet (MQLONG & inhibit);
 Provides a copy of the **inhibit get** value. It returns TRUE if successful.

MQLONG inhibitGet ();
 Returns the **inhibit get** value without any indication of possible errors.

ImqBoolean setInhibitGet (const MQLONG inhibit);
 Sets the **inhibit get** value. It returns TRUE if successful.

ImqBoolean inhibitPut (MQLONG & inhibit);
 Provides a copy of the **inhibit put** value. It returns TRUE if successful.

MQLONG inhibitPut ();
 Returns the **inhibit put** value without any indication of possible errors.

ImqBoolean setInhibitPut (const MQLONG inhibit);
 Sets the **inhibit put** value. It returns TRUE if successful.

ImqBoolean initiationQueueName (ImqString & name);
 Provides a copy of the **initiation queue name**. It returns TRUE if successful.

ImqString initiationQueueName ();
 Returns the **initiation queue name** without any indication of possible errors.

ImqBoolean maximumDepth (MQLONG & depth);
 Provides a copy of the **maximum depth**. It returns TRUE if successful.

MQLONG maximumDepth ();
 Returns the **maximum depth** without any indication of possible errors.

ImqBoolean maximumMessageLength (MQLONG & length);
 Provides a copy of the **maximum message length**. It returns TRUE if successful.

MQLONG maximumMessageLength ();
 Returns the **maximum message length** without any indication of possible errors.

ImqBoolean messageDeliverySequence (MQLONG & sequence);
 Provides a copy of the **message delivery sequence**. It returns TRUE if successful.

MQLONG messageDeliverySequence ();
 Returns the **message delivery sequence** value without any indication of possible errors.

ImqQueue * nextDistributedQueue () const ;
 Returns the **next distributed queue**.

ImqBoolean nonPersistentMessageClass (MQLONG & monq);
 Provides a copy of the non persistent message class value. It returns TRUE if successful.

MQLONG nonPersistentMessageClass ();
 Returns the non persistent message class value without any indication of possible errors.

ImqBoolean openInputCount (MQLONG & count);
 Provides a copy of the **open input count**. It returns TRUE if successful.

MQLONG openInputCount ();

Returns the **open input count** without any indication of possible errors.

ImqBoolean openOutputCount (MQLONG & count);

Provides a copy of the **open output count**. It returns TRUE if successful.

MQLONG openOutputCount ();

Returns the **open output count** without any indication of possible errors.

ImqQueue * previousDistributedQueue () const ;

Returns the **previous distributed queue**.

ImqBoolean processName (ImqString & name);

Provides a copy of the **process name**. It returns TRUE if successful.

ImqString processName ();

Returns the **process name** without any indication of possible errors.

ImqBoolean put (ImqMessage & msg);

Places a message onto the queue, using default put message options. Uses the ImqObject **openFor** method if necessary to ensure that the ImqObject **open options** include MQOO_OUTPUT.

This method returns TRUE if successful.

ImqBoolean put (ImqMessage & msg, ImqPutMessageOptions & pmo);

Places a message onto the queue, using the specified *pmo*. Uses the ImqObject **openFor** method as necessary to ensure that the ImqObject **open options** include MQOO_OUTPUT, and (if the *pmo options* include any of MQPMO_PASS_IDENTITY_CONTEXT, MQPMO_PASS_ALL_CONTEXT, MQPMO_SET_IDENTITY_CONTEXT, or MQPMO_SET_ALL_CONTEXT) corresponding MQOO_*_CONTEXT values.

This method returns TRUE if successful.

Note: If the *pmo* includes a **context reference**, the referenced object is opened, if necessary, to provide a context.

ImqBoolean queueAccounting (MQLONG & acctq);

Provides a copy of the queue accounting value. It returns TRUE if successful.

MQLONG queueAccounting ();

Returns the queue accounting value without any indication of possible errors.

ImqString queueManagerName () const ;

Returns the **queue manager name**.

ImqBoolean setQueueManagerName (const char * name);

Sets the **queue manager name**. The **queue manager name** can be set only while the ImqObject **open status** is FALSE. This method returns TRUE if successful.

ImqBoolean queueMonitoring (MQLONG & monq);

Provides a copy of the queue monitoring value. It returns TRUE if successful.

MQLONG queueMonitoring ();

Returns the queue monitoring value without any indication of possible errors.

ImqBoolean queueStatistics (MQLONG & statq);

Provides a copy of the queue statistics value. It returns TRUE if successful.

MQLONG queueStatistics ();

Returns the queue statistics value without any indication of possible errors.

ImqBoolean queueType (MQLONG & type);

Provides a copy of the **queue type** value. It returns TRUE if successful.

MQLONG queueType ();

Returns the **queue type** without any indication of possible errors.

ImqBoolean remoteQueueManagerName (ImqString & name);

Provides a copy of the **remote queue manager name**. It returns TRUE if successful.

ImqString remoteQueueManagerName ();

Returns the **remote queue manager name** without any indication of possible errors.

ImqBoolean remoteQueueName (ImqString & *name*);

Provides a copy of the **remote queue name**. It returns TRUE if successful.

ImqString remoteQueueName ();

Returns the **remote queue name** without any indication of possible errors.

ImqBoolean resolvedQueueManagerName(ImqString & *name*);

Provides a copy of the **resolved queue manager name**. It returns TRUE if successful.

Note: This method fails unless MQOO_RESOLVE_NAMES is among the ImqObject **open options**.

ImqString resolvedQueueManagerName();

Returns the **resolved queue manager name**, without any indication of possible errors.

ImqBoolean resolvedQueueName(ImqString & *name*);

Provides a copy of the **resolved queue name**. It returns TRUE if successful.

Note: This method fails unless MQOO_RESOLVE_NAMES is among the ImqObject **open options**.

ImqString resolvedQueueName();

Returns the **resolved queue name**, without any indication of possible errors.

ImqBoolean retentionInterval (MQLONG & *interval*);

Provides a copy of the **retention interval**. It returns TRUE if successful.

MQLONG retentionInterval ();

Returns the **retention interval** without any indication of possible errors.

ImqBoolean scope (MQLONG & *scope*);

Provides a copy of the **scope**. It returns TRUE if successful.

MQLONG scope ();

Returns the **scope** without any indication of possible errors.

ImqBoolean serviceInterval (MQLONG & *interval*);

Provides a copy of the **service interval**. It returns TRUE if successful.

MQLONG serviceInterval ();

Returns the **service interval** without any indication of possible errors.

ImqBoolean serviceIntervalEvent (MQLONG & *event*);

Provides a copy of the enablement state of the **service interval event**. It returns TRUE if successful.

MQLONG serviceIntervalEvent ();

Returns the enablement state of the **service interval event** without any indication of possible errors.

ImqBoolean shareability (MQLONG & *shareability*);

Provides a copy of the **shareability** value. It returns TRUE if successful.

MQLONG shareability ();

Returns the **shareability** value without any indication of possible errors.

ImqBoolean storageClass(ImqString & *class*);

Provides a copy of the **storage class**. It returns TRUE if successful.

ImqString storageClass();

Returns the **storage class** without any indication of possible errors.

ImqBoolean transmissionQueueName (ImqString & *name*);

Provides a copy of the **transmission queue name**. It returns TRUE if successful.

ImqString transmissionQueueName ();

Returns the **transmission queue name** without any indication of possible errors.

ImqBoolean triggerControl (MQLONG & *control*);

Provides a copy of the **trigger control** value. It returns TRUE if successful.

MQLONG triggerControl ();

Returns the **trigger control** value without any indication of possible errors.

ImqBoolean setTriggerControl (const MQLONG *control*);

Sets the **trigger control** value. It returns TRUE if successful.

ImqBoolean triggerData (ImqString & *data*);

Provides a copy of the **trigger data**. It returns TRUE if successful.

ImqString triggerData ();

Returns a copy of the **trigger data** without any indication of possible errors.

ImqBoolean setTriggerData (const char * *data*);

Sets the **trigger data**. It returns TRUE if successful.

ImqBoolean triggerDepth (MQLONG & *depth*);

Provides a copy of the **trigger depth**. It returns TRUE if successful.

MQLONG triggerDepth ();

Returns the **trigger depth** without any indication of possible errors.

ImqBoolean setTriggerDepth (const MQLONG *depth*);

Sets the **trigger depth**. It returns TRUE if successful.

ImqBoolean triggerMessagePriority (MQLONG & *priority*);

Provides a copy of the **trigger message priority**. It returns TRUE if successful.

MQLONG triggerMessagePriority ();

Returns the **trigger message priority** without any indication of possible errors.

ImqBoolean setTriggerMessagePriority (const MQLONG *priority*);

Sets the **trigger message priority**. It returns TRUE if successful.

ImqBoolean triggerType (MQLONG & *type*);

Provides a copy of the **trigger type**. It returns TRUE if successful.

MQLONG triggerType ();

Returns the **trigger type** without any indication of possible errors.

ImqBoolean setTriggerType (const MQLONG *type*);

Sets the **trigger type**. It returns TRUE if successful.

ImqBoolean usage (MQLONG & *usage*);

Provides a copy of the **usage** value. It returns TRUE if successful.

MQLONG usage ();

Returns the **usage** value without any indication of possible errors.

Object methods (protected)

void setNextDistributedQueue (ImqQueue * *queue* = 0);

Sets the **next distributed queue**.

Attention: Use this function only if you are sure it will not break the distributed queue list.

void setPreviousDistributedQueue (ImqQueue * *queue* = 0);

Sets the **previous distributed queue**.

Attention: Use this function only if you are sure it will not break the distributed queue list.

Reason codes

- MQRC_ATTRIBUTE_LOCKED
- MQRC_CONTEXT_OBJECT_NOT_VALID
- MQRC_CONTEXT_OPEN_ERROR
- MQRC_CURSOR_NOT_VALID
- MQRC_NO_BUFFER
- MQRC_REOPEN_EXCL_INPUT_ERROR
- MQRC_REOPEN_INQUIRE_ERROR

- MQRC_REOPEN_TEMPORARY_Q_ERROR
- (reason codes from MQGET)
- (reason codes from MQPUT)

ImqQueueManager C++ class

This class encapsulates a queue manager (an IBM MQ object of type MQOT_Q_MGR).

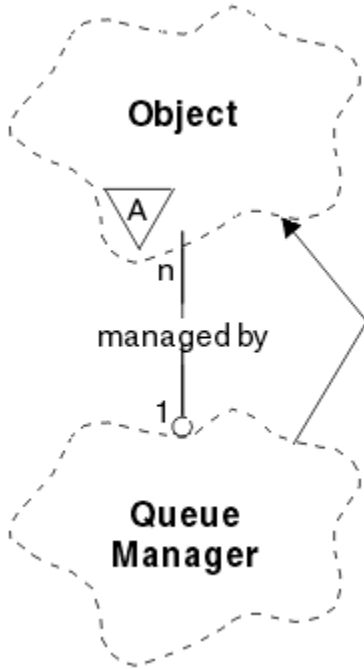


Figure 65. ImqQueueManager class

This class relates to the MQI calls listed in [“ImqQueueManager cross-reference”](#) on page 1832. Not all the listed methods are applicable to all platforms; see [ALTER QMGR](#) for more details.

- [“Class attributes”](#) on page 1900
- [“Object attributes”](#) on page 1901
- [“Constructors”](#) on page 1906
- [“Destructors”](#) on page 1906
- [“Class methods \(public\)”](#) on page 1906
- [“Object methods \(public\)”](#) on page 1906
- [“Object methods \(protected\)”](#) on page 1915
- [“Object data \(protected\)”](#) on page 1915
- [“Reason codes”](#) on page 1915

Class attributes

behavior

Controls the behavior of implicit connection and disconnection.

IMQ_EXPL_DISC_BACKOUT (0L)

An explicit call to the **disconnect** method implies backout. This attribute is mutually exclusive with IMQ_EXPL_DISC_COMMIT.

IMQ_EXPL_DISC_COMMIT (1L)

An explicit call to the **disconnect** method implies commit (the default). This attribute is mutually exclusive with IMQ_EXPL_DISC_BACKOUT.

IMQ_IMPL_CONN (2L)

Implicit connection is allowed (the default).

IMQ_IMPL_DISC_BACKOUT (0L)

An implicit call to the **disconnect** method, which can occur during object destruction, implies backout. This attribute is mutually exclusive with the IMQ_IMPL_DISC_COMMIT.

IMQ_IMPL_DISC_COMMIT (4L)

An implicit call to the **disconnect** method, which can occur during object destruction, implies commit (the default). This attribute is mutually exclusive with IMQ_IMPL_DISC_BACKOUT.

At IBM MQ V7.0 and above, C++ applications that make use of an implicit connection, need to specify IMQ_IMPL_CONN along with any other options provided in the `setBehavior()` method on an object of class `ImqQueueManager`. If your application does not use the `setBehavior()` method to explicitly set the behavior options, for example,

```
ImqQueueManager_object.setBehavior(IMQ_IMPL_DISC_COMMIT)
```

this change does not affect you since MQ_IMPL_CONN is enabled by default.

If your application explicitly sets the behavior options, for example,

```
ImqQueueManager_object.setBehavior(IMQ_IMPL_DISC_COMMIT)
```

you need to include IMQ_IMPL_CONN in the `setBehavior()` method as follows, to allow your application to complete an implicit connection:

```
ImqQueueManager_object.setBehavior(IMQ_IMPL_CONN | IMQ_IMPL_DISC_COMMIT)
```

Object attributes**accounting connections override**

Allows applications to override the setting of the MQI accounting and queue accounting values. This attribute is read-only.

accounting interval

How long before intermediate accounting records are written (in seconds). This attribute is read-only.

activity recording

Controls the generation of activity reports. This attribute is read-only.

adopt new mca check

The elements checked to determine if an MCA should be adopted when a new inbound channel is detected that has the same name as an MCA that is already active. This attribute is read-only.

adopt new mca type

Whether an orphaned instance of an MCA of a particular channel type should be restarted automatically when a new inbound channel request matching the adopt new mca check parameters is detected. This attribute is read-only.

authentication type

Indicates the type of authentication which is being performed.

authority event

Controls authority events. This attribute is read-only.

begin options

Options that apply to the **begin** method. The initial value is MQBO_NONE.

bridge event

Whether IMS bridge events are generated. This attribute is read-only.

channel auto definition

Channel auto definition value. This attribute is read-only.

channel auto definition event

Channel auto definition event value. This attribute is read-only.

channel auto definition exit

Channel auto definition exit name. This attribute is read-only.

channel event

Whether channel events are generated. This attribute is read-only.

channel initiator adapters

The number of adapter subtasks to use for processing IBM MQ calls. This attribute is read-only.

channel initiator control

Whether the Channel Initiator should be started automatically when the Queue Manager is started.
This attribute is read-only.

channel initiator dispatchers

The number of dispatchers to use for the channel initiator. This attribute is read-only.

channel initiator trace autostart

Whether channel initiator trace should start automatically or not. This attribute is read-only.

channel initiator trace table size

The size of the channel initiator's trace data space (in MB). This attribute is read-only.

channel monitoring

Controls the collection of online monitoring data for channels. This attribute is read-only.

channel reference

A reference to a channel definition for use during client connection. While connected, this attribute can be set to null, but cannot be changed to any other value. The initial value is null.

channel statistics

Controls the collection of statistics data for channels. This attribute is read-only.

character set

Coded character set identifier (CCSID). This attribute is read-only.

cluster sender monitoring

Controls the collection of online monitoring data for automatically-defined cluster sender channels.
This attribute is read-only.

cluster sender statistics

Controls the collection of statistics data for automatically defined cluster sender channels. This attribute is read-only.

cluster workload data

Cluster workload exit data. This attribute is read-only.

cluster workload exit

Cluster workload exit name. This attribute is read-only.

cluster workload length

Cluster workload length. This attribute is read-only.

cluster workload mru

Cluster workload most recently used channels value. This attribute is read-only.

cluster workload use queue

Cluster workload use queue value. This attribute is read-only.

command event

Whether command events are generated. This attribute is read-only.

command input queue name

System command input queue name. This attribute is read-only.

command level

Command level supported by the queue manager. This attribute is read-only.

command server control

Whether the Command Server should be started automatically when the Queue Manager is started.
This attribute is read-only.

connect options

Options that apply to the **connect** method. The initial value is MQCNO_NONE. The following additional values may be possible, depending on platform:

- MQCNO_STANDARD_BINDING
- MQCNO_FASTPATH_BINDING
- MQCNO_HANDLE_SHARE_NONE
- MQCNO_HANDLE_SHARE_BLOCK
- MQCNO_HANDLE_SHARE_NO_BLOCK
- MQCNO_SERIALIZE_CONN_TAG_Q_MGR
- MQCNO_SERIALIZE_CONN_TAG_QSG
- MQCNO_RESTRICT_CONN_TAG_Q_MGR
- MQCNO_RESTRICT_CONN_TAG_QSG

connection id

A unique identifier that allows MQ to reliably identify an application.

connection status

TRUE when connected to the queue manager. This attribute is read-only.

connection tag

A tag to be associated with a connection. This attribute can only be set when not connected. The initial value is null.

cryptographic hardware

Configuration details for cryptographic hardware. For MQ MQI client connections.

dead-letter queue name

Name of the dead-letter queue. This attribute is read-only.

default transmission queue name

Default transmission queue name. This attribute is read-only.

distribution lists

Capability of the queue manager to support distribution lists.

dns group

The name of the group that the TCP listener that handles inbound transmissions for the queue-sharing group should join when using Workload Manager Dynamic Domain Name Services support. This attribute is read-only.

dns wlm

Whether the TCP listener that handles inbound transmissions for the queue-sharing group should register with Workload Manager for Dynamic Domain Name Services. This attribute is read-only.

first authentication record

The first of one or more objects of class ImqAuthenticationRecord, in no particular order, in which the ImqAuthenticationRecord connection reference addresses this object. For MQ MQI client connections.

first managed object

The first of one or more objects of class ImqObject, in no particular order, in which the ImqObject **connection reference** addresses this object. The initial value is zero.

inhibit event

Controls inhibit events. This attribute is read-only.

ip address version

Which IP protocol (IPv4 or IPv6) to use for a channel connection. This attribute is read-only.

key repository

Location of the key database file in which keys and certificates are stored. For IBM MQ MQI client connections.

key reset count

The number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated. This attribute applies only to client connections using MQCONN. See also [ssl key reset count](#).

listener timer

The time interval (in seconds) between attempts by IBM MQ to restart the listener if there has been an APPC or TCP/IP failure. This attribute is read-only.

local event

Controls local events. This attribute is read-only.

logger event

Controls whether recovery log events are generated. This attribute is read-only.

lu group name

The generic LU name that the LU 6.2 listener that handles inbound transmissions for the queue-sharing group should use. This attribute is read-only.

lu name

The name of the LU to use for outbound LU 6.2 transmissions. This attribute is read-only.

lu62 arm suffix

The suffix of the SYS1.PARMLIB member APPCPMxx, that nominates the LUADD for this channel initiator. This attribute is read-only.

lu62 channels

The maximum number of channels that can be current or clients that can be connected, that use the LU 6.2 transmission protocol. This attribute is read-only.

maximum active channels

The maximum number of channels that can be active at any time. This attribute is read-only.

maximum channels

The maximum number of channels that can be current (including server-connection channels with connected clients). This attribute is read-only.

maximum handles

Maximum number of handles. This attribute is read-only.

maximum message length

Maximum possible length for any message on any queue managed by this queue manager. This attribute is read-only.

maximum priority

Maximum message priority. This attribute is read-only.

maximum uncommitted messages

Maximum number of uncommitted messages within a unit of work. This attribute is read-only.

mqi accounting

Controls the collection of accounting information for MQI data. This attribute is read-only.

mqi statistics

Controls the collection of statistics monitoring information for the queue manager. This attribute is read-only.

outbound port maximum

The higher end of the range of port numbers to be used when binding outgoing channels. This attribute is read-only.

outbound port minimum

The lower end of the range of port numbers to be used when binding outgoing channels. This attribute is read-only.

password

password associated with user ID

performance event

Controls performance events. This attribute is read-only.

platform

Platform on which the queue manager resides. This attribute is read-only.

queue accounting

Controls the collection of accounting information for queues. This attribute is read-only.

queue monitoring

Controls the collection of online monitoring data for queues. This attribute is read-only.

queue statistics

Controls the collection of statistics data for queues. This attribute is read-only.

receive timeout

Approximately how long a TCP/IP message channel will wait to receive data, including heartbeats, from its partner, before returning to the inactive state. This attribute is read-only.

receive timeout minimum

The minimum time that a TCP/IP channel will wait to receive data, including heartbeats, from its partner, before returning to the inactive state. This attribute is read-only.

receive timeout type

A qualifier applied to **receive timeout**. This attribute is read-only.

remote event

Controls remote events. This attribute is read-only.

repository name

Repository name. This attribute is read-only.

repository namelist

Repository namelist name. This attribute is read-only.

shared queue manager name

Whether MQOPENS of a shared queue where the ObjectQMGrName is another queue manager in the queue-sharing group should resolve to an open of the shared queue on the local queue manager. This attribute is read-only.

ssl event

Whether SSL events are generated. This attribute is read-only.

ssl FIPS required

Whether only FIPS-certified algorithms should be used if the cryptography is executed in IBM MQ software. This attribute is read-only.

ssl key reset count

The number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated. This attribute is read-only.

start-stop event

Controls start-stop events. This attribute is read-only.


statistics interval

How often statistics monitoring data is written to the monitoring queue. This attribute is read-only.

syncpoint availability

Availability of syncpoint participation. This attribute is read-only.

Note: Queue manager-coordinated global units of work are not supported on the IBM i platform.

 You can program a unit of work, externally coordinated by IBM i, using the `_Rcommit` and `_Rback` native system calls. Start this type of unit of work by starting the IBM MQ application under job-level commitment control using the `STRCMTCTL` command. See [Interfaces to the IBM i external syncpoint manager](#) for further details. **Backout** and **commit** are supported on the IBM i platform for local units of work coordinated by a queue manager.

tcp channels

The maximum number of channels that can be current or clients that can be connected, that use the TCP/IP transmission protocol. This attribute is read-only.

tcp keepalive

Whether the TCP KEEPALIVE facility is to be used to check that the other end of the connection is still available. This attribute is read-only.

tcp name

The name of either the sole or default TCP/IP system to be used, depending on the value of **tcp stack type**. This attribute is read-only.

tcp stack type

Whether the channel initiator is permitted to only use the TCP/IP address space specified in **tcp name** or can bind to any selected TCP/IP address. This attribute is read-only.

trace route recording

Controls the recording of route tracing information. This attribute is read-only.

trigger interval

Trigger interval. This attribute is read-only.

user id

On UNIX and Linux platforms, the application's real user ID. On Windows platforms, the application's user ID.

Constructors**ImqQueueManager();**

The default constructor.

ImqQueueManager(const ImqQueueManager & *manager*);

The copy constructor. The **connection status** will be FALSE.

ImqQueueManager(const char * *name*);

Sets the ImqObject **name** to *name*.

Destructors

When an ImqQueueManager object is destroyed, it is automatically disconnected.

Class methods (public)**static MQLONG behavior();**

Returns the **behavior**.

void setBehavior(const MQLONG *behavior* = 0);

Sets the **behavior**.

Object methods (public)**void operator = (const ImqQueueManager & *mgr*);**

Disconnects if necessary, and copies instance data from *mgr*. The **connection status** is be FALSE.

ImqBoolean accountingConnOverride (MQLONG & *statint*);

Provides a copy of the accounting connections override value. It returns TRUE if successful.

MQLONG accountingConnOverride ();

Returns the accounting connections override value without any indication of possible errors.

ImqBoolean accountingInterval (MQLONG & *statint*);

Provides a copy of the accounting interval value. It returns TRUE if successful.

MQLONG accountingInterval ();

Returns the accounting interval value without any indication of possible errors.

ImqBoolean activityRecording (MQLONG & *rec*);

Provides a copy of the activity recording value. It returns TRUE if successful.

MQLONG activityRecording ();

Returns the activity recording value without any indication of possible errors.

ImqBoolean adoptNewMCACheck (MQLONG & check);

Provides a copy of the adopt new MCA check value. It returns TRUE if successful.

MQLONG adoptNewMCACheck ();

Returns the adopt new MCA check value without any indication of possible errors.

ImqBoolean adoptNewMCAType (MQLONG & type);

Provides a copy of the adopt new MCA type. It returns TRUE if successful.

MQLONG adoptNewMCAType ();

Returns the adopt new MCA type without any indication of possible errors.

QLONG authenticationType () const;

Returns the authentication type.

void setAuthenticationType (const MQLONG type = MQCSP_AUTH_NONE);

Sets the authentication type.

ImqBoolean authorityEvent(MQLONG & event);

Provides a copy of the enablement state of the **authority event**. It returns TRUE if successful.

MQLONG authorityEvent();

Returns the enablement state of the **authority event** without any indication of possible errors.

ImqBoolean backout();

Backs out uncommitted changes. It returns TRUE if successful.

ImqBoolean begin();

Begins a unit of work. The **begin options** affect the behavior of this method. It returns TRUE if successful, but it also returns TRUE even if the underlying MQBEGIN call returns MQRC_NO_EXTERNAL_PARTICIPANTS or MQRC_PARTICIPANT_NOT_AVAILABLE (which are both associated with MQCC_WARNING).

MQLONG beginOptions() const ;

Returns the **begin options**.

void setBeginOptions(const MQLONG options = MQBO_NONE);

Sets the **begin options**.

ImqBoolean bridgeEvent (MQLONG & event);

Provides a copy of the bridge event value. It returns TRUE if successful.

MQLONG bridgeEvent ();

Returns the bridge event value without any indication of possible errors.

ImqBoolean channelAutoDefinition(MQLONG & value);

Provides a copy of the **channel auto definition** value. It returns TRUE if successful.

MQLONG channelAutoDefinition();

Returns the **channel auto definition** value without any indication of possible errors.

ImqBoolean channelAutoDefinitionEvent(MQLONG & value);

Provides a copy of the **channel auto definition event** value. It returns TRUE if successful.

MQLONG channelAutoDefinitionEvent();

Returns the **channel auto definition event** value without any indication of possible errors.

ImqBoolean channelAutoDefinitionExit(ImqString & name);

Provides a copy of the **channel auto definition exit** name. It returns TRUE if successful.

ImqString channelAutoDefinitionExit();

Returns the **channel auto definition exit** name without any indication of possible errors.

ImqBoolean channelEvent (MQLONG & event);

Provides a copy of the channel event value. It returns TRUE if successful.

MQLONG channelEvent();

Returns the channel event value without any indication of possible errors.

MQLONG channelInitiatorAdapters ();

Returns the channel initiator adapters value without any indication of possible errors.

ImqBoolean channelInitiatorAdapters (MQLONG & adapters);
Provides a copy of the channel initiator adapters value. It returns TRUE if successful.

MQLONG channelInitiatorControl ();
Returns the channel initiator startup value without any indication of possible errors.

ImqBoolean channelInitiatorControl (MQLONG & init);
Provides a copy of the channel initiator control startup value. It returns TRUE if successful.

MQLONG channelInitiatorDispatchers ();
Returns the channel initiator dispatchers value without any indication of possible errors.

ImqBoolean channelInitiatorDispatchers (MQLONG & dispatchers);
Provides a copy of the channel initiator dispatchers value. It returns TRUE if successful.

MQLONG channelInitiatorTraceAutoStart ();
Returns the channel initiator trace auto start value without any indication of possible errors.

ImqBoolean channelInitiatorTraceAutoStart (MQLONG & auto);
Provides a copy of the channel initiator trace auto start value. It returns TRUE if successful.

MQLONG channelInitiatorTraceTableSize ();
Returns the channel initiator trace table size value without any indication of possible errors.

ImqBoolean channelInitiatorTraceTableSize (MQLONG & size);
Provides a copy of the channel initiator trace table size value. It returns TRUE if successful.

ImqBoolean channelMonitoring (MQLONG & monchl);
Provides a copy of the channel monitoring value. It returns TRUE if successful.

MQLONG channelMonitoring ();
Returns the channel monitoring value without any indication of possible errors.

ImqBoolean channelReference(ImqChannel * & pchannel);
Provides a copy of the **channel reference**. If the **channel reference** is invalid, sets *pchannel* to null. This method returns TRUE if successful.

ImqChannel * channelReference();
Returns the **channel reference** without any indication of possible errors.

ImqBoolean setChannelReference(ImqChannel & channel);
Sets the **channel reference**. This method returns TRUE if successful.

ImqBoolean setChannelReference(ImqChannel * channel = 0);
Sets or resets the **channel reference**. This method returns TRUE if successful.

ImqBoolean channelStatistics (MQLONG & statchl);
Provides a copy of the channel statistics value. It returns TRUE if successful.

MQLONG channelStatistics ();
Returns the channel statistics value without any indication of possible errors.

ImqBoolean characterSet(MQLONG & ccscid);
Provides a copy of the **character set**. It returns TRUE if successful.

MQLONG characterSet();
Returns a copy of the **character set**, without any indication of possible errors.

MQLONG clientSslKeyResetCount () const;
Returns the SSL key reset count value used on client connections.

void setClientSslKeyResetCount(const MQLONG count);
Sets the SSL key reset count used on client connections.

ImqBoolean clusterSenderMonitoring (MQLONG & monacsl);
Provides a copy of the cluster sender monitoring default value. It returns TRUE if successful.

MQLONG clusterSenderMonitoring ();
Returns the cluster sender monitoring default value without any indication of possible errors.

ImqBoolean clusterSenderStatistics (MQLONG & statacls);
Provides a copy of the cluster sender statistics value. It returns TRUE if successful.

MQLONG clusterSenderStatistics ();

Returns the cluster sender statistics value without any indication of possible errors.

ImqBoolean clusterWorkloadData(ImqString & data);

Provides a copy of the **cluster workload exit data**. It returns TRUE if successful.

ImqString clusterWorkloadData();

Returns the **cluster workload exit data** without any indication of possible errors.

ImqBoolean clusterWorkloadExit(ImqString & name);

Provides a copy of the **cluster workload exit name**. It returns TRUE if successful.

ImqString clusterWorkloadExit();

Returns the **cluster workload exit name** without any indication of possible errors.

ImqBoolean clusterWorkloadLength(MQLONG & length);

Provides a copy of the **cluster workload length**. It returns TRUE if successful.

MQLONG clusterWorkloadLength();

Returns the **cluster workload length** without any indication of possible errors.

ImqBoolean clusterWorkLoadMRU (MQLONG & mru);

Provides a copy of the cluster workload most recently used channels value. It returns TRUE if successful.

MQLONG clusterWorkLoadMRU ();

Returns the cluster workload most recently used channels value without any indication of possible errors.

ImqBoolean clusterWorkLoadUseQ (MQLONG & useq);

Provides a copy of the cluster workload use queue value. It returns TRUE if successful.

MQLONG clusterWorkLoadUseQ ();

Returns the cluster workload use queue value without any indication of possible errors.

ImqBoolean commandEvent (MQLONG & event);

Provides a copy of the command event value. It returns TRUE if successful.

MQLONG commandEvent ();

Returns the command event value without any indication of possible errors.

ImqBoolean commandInputQueueName(ImqString & name);

Provides a copy of the **command input queue name**. It returns TRUE if successful.

ImqString commandInputQueueName();

Returns the **command input queue name** without any indication of possible errors.

ImqBoolean commandLevel(MQLONG & level);

Provides a copy of the **command level**. It returns TRUE if successful.

MQLONG commandLevel();

Returns the **command level** without any indication of possible errors.

MQLONG commandServerControl ();

Returns the command server startup value without any indication of possible errors.

ImqBoolean commandServerControl (MQLONG & server);

Provides a copy of the command server control startup value. It returns TRUE if successful.

ImqBoolean commit();

Commits uncommitted changes. It returns TRUE if successful.

ImqBoolean connect();

Connects to the queue manager with the given ImqObject **name**, the default being the local queue manager. If you want to connect to a specific queue manager, use the ImqObject **setName** method before connection. If there is a **channel reference**, it is used to pass information about the channel definition to MQCONN in an MQCD. The ChannelType in the MQCD is set to MQCHT_CLNTCONN. **channel reference** information, which is only meaningful for client connections, is ignored for server connections. The **connect options** affect the behavior of this method. This method sets the **connection status** to TRUE if successful. It returns the new connection status.

If there is a first authentication record, the chain of authentication records is used to authenticate digital certificates for secure client channels.

You can connect more than one ImqQueueManager object to the same queue manager. All use the same MQHCONN connection handle and share UOW functionality for the connection associated with the thread. The first ImqQueueManager to connect obtains the MQHCONN handle. The last ImqQueueManager to disconnect performs the MQDISC.

For a multithreaded program, it is recommended that a separate ImqQueueManager object is used for each thread.

ImqBinary connectionId () const ;

Returns the **connection ID**.

ImqBinary connectionTag () const ;

Returns the **connection tag**.

ImqBoolean setConnectionTag (const MQBYTE128 tag = 0);

Sets the **connection tag**. If *tag* is zero, clears the **connection tag**. This method returns TRUE if successful.

ImqBoolean setConnectionTag (const ImqBinary & tag);

Sets the **connection tag**. The **data length** of *tag* must be either zero (to clear the **connection tag**) or MQ_CONN_TAG_LENGTH. This method returns TRUE if successful.

MQLONG connectOptions() const ;

Returns the **connect options**.

void setConnectOptions(const MQLONG options = MQCNO_NONE);

Sets the **connect options**.

ImqBoolean connectionStatus() const ;

Returns the **connection status**.

ImqString cryptographicHardware ();

Returns the **cryptographic hardware**.

ImqBoolean setCryptographicHardware (const char * hardware = 0);

Sets the **cryptographic hardware**. This method returns TRUE if successful.

ImqBoolean deadLetterQueueName(ImqString & name);

Provides a copy of the **dead-letter queue name**. It returns TRUE if successful.

ImqString deadLetterQueueName();

Returns a copy of the **dead-letter queue name**, without any indication of possible errors.

ImqBoolean defaultTransmissionQueueName(ImqString & name);

Provides a copy of the **default transmission queue name**. It returns TRUE if successful.

ImqString defaultTransmissionQueueName();

Returns the **default transmission queue name** without any indication of possible errors.

ImqBoolean disconnect();

Disconnects from the queue manager and sets the **connection status** to FALSE. Closes all ImqProcess and ImqQueue objects associated with this object, and severs their **connection reference** before disconnection. If more than one ImqQueueManager object is connected to the same queue manager, only the last to disconnect performs a physical disconnection; others perform a logical disconnection. Uncommitted changes are committed on physical disconnection only.

This method returns TRUE if successful. If it is called when there is no existing connection, the return code is also true.

ImqBoolean distributionLists(MQLONG & support);

Provides a copy of the **distribution lists** value. It returns TRUE if successful.

MQLONG distributionLists();

Returns the **distribution lists** value without any indication of possible errors.

ImqBoolean dnsGroup (ImqString & group);

Provides a copy of the DNS group name. It returns TRUE if successful.

ImqString dnsGroup ();
Returns the DNS group name without any indication of possible errors.

ImqBoolean dnsWlm (MQLONG & wlm);
Provides a copy of the DNS WLM value. It returns TRUE if successful.

MQLONG dnsWlm ();
Returns the DNS WLM value without any indication of possible errors.

ImqAuthenticationRecord * firstAuthenticationRecord () const ;
Returns the **first authentication record**.

void setFirstAuthenticationRecord (const ImqAuthenticationRecord * air = 0);
Sets the **first authentication record**.

ImqObject * firstManagedObject () const ;
Returns the **first managed object**.

ImqBoolean inhibitEvent(MQLONG & event);
Provides a copy of the enablement state of the **inhibit event**. It returns TRUE if successful.

MQLONG inhibitEvent();
Returns the enablement state of the **inhibit event** without any indication of possible errors.

ImqBoolean ipAddressVersion (MQLONG & version);
Provides a copy of the IP address version value. It returns TRUE if successful.

MQLONG ipAddressVersion ();
Returns the IP address version value without any indication of possible errors.

ImqBoolean keepAlive (MQLONG & keepalive);
Provides a copy of the keep alive value. It returns TRUE if successful.

MQLONG keepAlive ();
Returns the keep alive value without any indication of possible errors.

ImqString keyRepository ();
Returns the **key repository**.

ImqBoolean setKeyRepository (const char * repository = 0);
Sets the **key repository**. It returns TRUE if successful.

ImqBoolean listenerTimer (MQLONG & timer);
Provides a copy of the listener timer value. It returns TRUE if successful.

MQLONG listenerTimer ();
Returns the listener timer value without any indication of possible errors.

ImqBoolean localEvent(MQLONG & event);
Provides a copy of the enablement state of the **local event**. It returns TRUE if successful.

MQLONG localEvent();
Returns the enablement state of the **local event** without any indication of possible errors.

ImqBoolean loggerEvent (MQLONG & count);
Provides a copy of the logger event value. It returns TRUE if successful.

MQLONG loggerEvent ();
Returns the logger event value without any indication of possible errors.

ImqBoolean luGroupName (ImqString & name);
Provides a copy of the LU group name. It returns TRUE if successful.

ImqString luGroupName ();
Returns the LU group name without any indication of possible errors.

ImqBoolean lu62ARMSuffix (ImqString & suffix);
Provides a copy of the LU62 ARM suffix. It returns TRUE if successful.

ImqString lu62ARMSuffix ();
Returns the LU62 ARM suffix without any indication of possible errors.

ImqBoolean luName (ImqString & name);
Provides a copy of the LU name. It returns TRUE if successful.

ImqString luName ();

Returns the LU name without any indication of possible errors.

ImqBoolean maximumActiveChannels (MQLONG & channels);

Provides a copy of the maximum active channels value. It returns TRUE if successful.

MQLONG maximumActiveChannels ();

Returns the maximum active channels value without any indication of possible errors.

ImqBoolean maximumCurrentChannels (MQLONG & channels);

Provides a copy of the maximum current channels value. It returns TRUE if successful.

MQLONG maximumCurrentChannels ();

Returns the maximum current channels value without any indication of possible errors.

ImqBoolean maximumHandles(MQLONG & number);

Provides a copy of the **maximum handles**. It returns TRUE if successful.

MQLONG maximumHandles();

Returns the **maximum handles** without any indication of possible errors.

ImqBoolean maximumLu62Channels (MQLONG & channels);

Provides a copy of the maximum LU62 channels value. It returns TRUE if successful.

MQLONG maximumLu62Channels ();

Returns the maximum LU62 channels value without any indication of possible errors

ImqBoolean maximumMessageLength(MQLONG & length);

Provides a copy of the **maximum message length**. It returns TRUE if successful.

MQLONG maximumMessageLength();

Returns the **maximum message length** without any indication of possible errors.

ImqBoolean maximumPriority(MQLONG & priority);

Provides a copy of the **maximum priority**. It returns TRUE if successful.

MQLONG maximumPriority();

Returns a copy of the **maximum priority**, without any indication of possible errors.

ImqBoolean maximumTcpChannels (MQLONG & channels);

Provides a copy of the maximum TCP channels value. It returns TRUE if successful.

MQLONG maximumTcpChannels ();

Returns the maximum TCP channels value without any indication of possible errors.

ImqBoolean maximumUncommittedMessages(MQLONG & number);

Provides a copy of the **maximum uncommitted messages**. It returns TRUE if successful.

MQLONG maximumUncommittedMessages();

Returns the **maximum uncommitted messages** without any indication of possible errors.

ImqBoolean mqiAccounting (MQLONG & statint);

Provides a copy of the MQI accounting value. It returns TRUE if successful.

MQLONG mqiAccounting ();

Returns the MQI accounting value without any indication of possible errors.

ImqBoolean mqiStatistics (MQLONG & statmqi);

Provides a copy of the MQI statistics value. It returns TRUE if successful.

MQLONG mqiStatistics ();

Returns the MQI statistics value without any indication of possible errors.

ImqBoolean outboundPortMax (MQLONG & max);

Provides a copy of the maximum outbound port value. It returns TRUE if successful.

MQLONG outboundPortMax ();

Returns the maximum outbound port value without any indication of possible errors.

ImqBoolean outboundPortMin (MQLONG & min);

Provides a copy of the minimum outbound port value. It returns TRUE if successful.

MQLONG outboundPortMin ();

Returns the minimum outbound port value without any indication of possible errors.

ImqBinary password () const;

Returns the password used on client connections.

ImqBoolean setPassword (const ImqString & password);

Sets the password used on client connections.

ImqBoolean setPassword (const char * = 0 password);

Sets the password used on client connections.

ImqBoolean setPassword (const ImqBinary & password);

Sets the password used on client connections.

ImqBoolean performanceEvent(MQLONG & event);

Provides a copy of the enablement state of the **performance event**. It returns TRUE if successful.

MQLONG performanceEvent();

Returns the enablement state of the **performance event** without any indication of possible errors.

ImqBoolean platform(MQLONG & platform);

Provides a copy of the **platform**. It returns TRUE if successful.

MQLONG platform();

Returns the **platform** without any indication of possible errors.

ImqBoolean queueAccounting (MQLONG & acctq);

Provides a copy of the queue accounting value. It returns TRUE if successful.

MQLONG queueAccounting ();

Returns the queue accounting value without any indication of possible errors.

ImqBoolean queueMonitoring (MQLONG & monq);

Provides a copy of the queue monitoring value. It returns TRUE if successful.

MQLONG queueMonitoring ();

Returns the queue monitoring value without any indication of possible errors.

ImqBoolean queueStatistics (MQLONG & statq);

Provides a copy of the queue statistics value. It returns TRUE if successful.

MQLONG queueStatistics ();

Returns the queue statistics value without any indication of possible errors.

ImqBoolean receiveTimeout (MQLONG & timeout);

Provides a copy of the receive timeout value. It returns TRUE if successful.

MQLONG receiveTimeout ();

Returns the receive timeout value without any indication of possible errors.

ImqBoolean receiveTimeoutMin (MQLONG & min);

Provides a copy of the minimum receive timeout value. It returns TRUE if successful.

MQLONG receiveTimeoutMin ();

Returns the minimum receive timeout value without any indication of possible errors.

ImqBoolean receiveTimeoutType (MQLONG & type);

Provides a copy of the receive timeout type. It returns TRUE if successful.

MQLONG receiveTimeoutType ();

Returns the receive timeout type without any indication of possible errors.

ImqBoolean remoteEvent(MQLONG & event);

Provides a copy of the enablement state of the **remote event**. It returns TRUE if successful.

MQLONG remoteEvent();

Returns the enablement state of the **remote event** without any indication of possible errors.

ImqBoolean repositoryName(ImqString & name);

Provides a copy of the **repository name**. It returns TRUE if successful.

ImqString repositoryName();

Returns the **repository name** without any indication of possible errors.

ImqBoolean repositoryNameListName(ImqString & name);

Provides a copy of the **repository namelist name**. It returns TRUE if successful.

ImqString repositoryNamelistName();

Returns a copy of the **repository namelist name** without any indication of possible errors.

ImqBoolean sharedQueueQueueManagerName (MQLONG & name);

Provides a copy of the shared queue queue manager name value. It returns TRUE if successful.

MQLONG sharedQueueQueueManagerName ();

Returns the shared queue queue manager name value without any indication of possible errors.

ImqBoolean sslEvent (MQLONG & event);

Provides a copy of the SSL event value. It returns TRUE if successful.

MQLONG sslEvent ();

Returns the SSL event value without any indication of possible errors.

ImqBoolean sslFips (MQLONG & sslfips);

Provides a copy of the SSL FIPS value. It returns TRUE if successful.

MQLONG sslFips ();

Returns the SSL FIPS value without any indication of possible errors.

ImqBoolean sslKeyResetCount (MQLONG & count);

Provides a copy of the SSL key reset count value. It returns TRUE if successful.

MQLONG sslKeyResetCount ();

Returns the SSL key reset count value without any indication of possible errors.

ImqBoolean startStopEvent(MQLONG & event);

Provides a copy of the enablement state of the **start-stop event**. It returns TRUE if successful.

MQLONG startStopEvent();

Returns the enablement state of the **start-stop event** without any indication of possible errors.

ImqBoolean statisticsInterval (MQLONG & statint);

Provides a copy of the statistics interval value. It returns TRUE if successful.

MQLONG statisticsInterval ();

Returns the statistics interval value without any indication of possible errors.

ImqBoolean syncPointAvailability(MQLONG & sync);

Provides a copy of the **syncpoint availability** value. It returns TRUE if successful.

MQLONG syncPointAvailability();

Returns a copy of the **syncpoint availability** value, without any indication of possible errors.

ImqBoolean tcpName (ImqString & name);

Provides a copy of the TCP system name. It returns TRUE if successful.

ImqString tcpName ();

Returns the TCP system name without any indication of possible errors.

ImqBoolean tcpStackType (MQLONG & type);

Provides a copy of the TCP stack type. It returns TRUE if successful.

MQLONG tcpStackType ();

Returns the TCP stack type without any indication of possible errors.

ImqBoolean traceRouteRecording (MQLONG & routerec);

Provides a copy of the trace route recording value. It returns TRUE if successful.

MQLONG traceRouteRecording ();

Returns the trace route recording value without any indication of possible errors.

ImqBoolean triggerInterval(MQLONG & interval);

Provides a copy of the **trigger interval**. It returns TRUE if successful.

MQLONG triggerInterval();

Returns the **trigger interval** without any indication of possible errors.

ImqBinary userId () const;

Returns the user ID used on client connections.

ImqBoolean setUserId (const ImqString & id);

Sets the user ID used on client connections.

ImqBoolean setUserId (const char * = 0 id);

Sets the user ID used on client connections.

ImqBoolean setUserId (const ImqBinary & id);

Sets the user ID used on client connections.

Object methods (protected)

void setFirstManagedObject (const ImqObject * object = 0);

Sets the **first managed object**.

Object data (protected)

MQHCONN ohconn

The IBM MQ connection handle (meaningful only while the **connection status** is TRUE).

Reason codes

- MQRC_ATTRIBUTE_LOCKED
- MQRC_ENVIRONMENT_ERROR
- MQRC_FUNCTION_NOT_SUPPORTED
- MQRC_REFERENCE_ERROR
- (reason codes for MQBACK)
- (reason codes for MQBEGIN)
- (reason codes for MQCMIT)
- (reason codes for MQCONN)
- (reason codes for MQCONNX)
- (reason codes for MQDISC)
- (reason codes for MQCONN)

ImqReferenceHeader C++ class

This class encapsulates features of the MQRMH data structure.

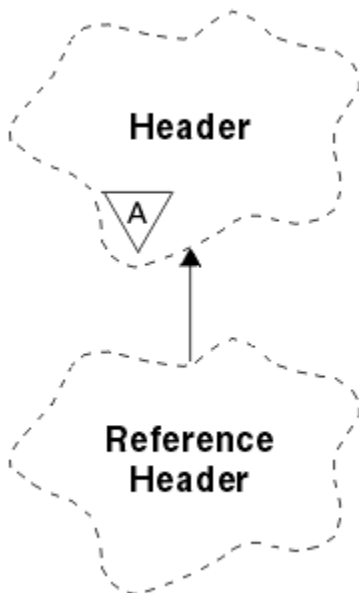


Figure 66. *ImqReferenceHeader* class

This class relates to the MQI calls listed in [“ImqReferenceHeader cross-reference”](#) on page 1836.

- [“Object attributes” on page 1916](#)
- [“Constructors” on page 1916](#)
- [“Overloaded ImqItem methods” on page 1916](#)
- [“Object methods \(public\)” on page 1917](#)
- [“Object data \(protected\)” on page 1917](#)
- [“Reason codes” on page 1918](#)

Object attributes

destination environment

Environment for the destination. The initial value is a null string.

destination name

Name of the data destination. The initial value is a null string.

instance id

Instance identifier. A binary value (MQBYTE24) of length MQ_OBJECT_INSTANCE_ID_LENGTH. The initial value is MQOII_NONE.

logical length

Logical, or intended, length of message data that follows this header. The initial value is zero.

logical offset

Logical offset for the message data that follows, to be interpreted in the context of the data as a whole, at the ultimate destination. The initial value is zero.

logical offset 2

High-order extension to the **logical offset**. The initial value is zero.

reference type

Reference type. The initial value is a null string.

source environment

Environment for the source. The initial value is a null string.

source name

Name of the data source. The initial value is a null string.

Constructors

ImqReferenceHeader();

The default constructor.

ImqReferenceHeader(const ImqReferenceHeader & header);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & msg);

Inserts an MQRMH data structure into the message buffer at the beginning, moving existing message data further along, and sets the *msg* **format** to MQFMT_REF_MSG_HEADER.

See the ImqHeader class method description on [“ImqHeader C++ class” on page 1864](#) for further details.

virtual ImqBoolean pasteIn (ImqMessage & msg);

Reads an MQRMH data structure from the message buffer.

To be successful, the ImqMessage **format** must be MQFMT_REF_MSG_HEADER.

See the ImqHeader class method description on [“ImqHeader C++ class” on page 1864](#) for further details.

Object methods (public)

void operator = (const ImqReferenceHeader & *header*);

Copies instance data from *header*, replacing the existing instance data.

ImqString destinationEnvironment () const ;

Returns a copy of the **destination environment**.

void setDestinationEnvironment (const char * *environment* = 0);

Sets the **destination environment**.

ImqString destinationName () const ;

Returns a copy of the **destination name**.

void setDestinationName (const char * *name* = 0);

Sets the **destination name**.

ImqBinary instanceId () const ;

Returns a copy of the **instance id**.

ImqBoolean setInstanceId (const ImqBinary & *id*);

Sets the **instance id**. The **data length** of *token* must be either 0 or MQ_OBJECT_INSTANCE_ID_LENGTH. This method returns TRUE if successful.

void setInstanceId (const MQBYTE24 *id* = 0);

Sets the **instance id**. *id* can be zero, which is the same as specifying MQOII_NONE. If *id* is nonzero, it must address MQ_OBJECT_INSTANCE_ID_LENGTH bytes of binary data. When using pre-defined values such as MQOII_NONE, you might need to make a cast to ensure a signature match, for example (MQBYTE *)MQOII_NONE.

MQLONG logicalLength () const ;

Returns the **logical length**.

void setLogicalLength (const MQLONG *length*);

Sets the **logical length**.

MQLONG logicalOffset () const ;

Returns the **logical offset**.

void setLogicalOffset (const MQLONG *offset*);

Sets the **logical offset**.

MQLONG logicalOffset2 () const ;

Returns the **logical offset 2**.

void setLogicalOffset2 (const MQLONG *offset*);

Sets the **logical offset 2**.

ImqString referenceType () const ;

Returns a copy of the **reference type**.

void setReferenceType (const char * *name* = 0);

Sets the **reference type**.

ImqString sourceEnvironment () const ;

Returns a copy of the **source environment**.

void setSourceEnvironment (const char * *environment* = 0);

Sets the **source environment**.

ImqString sourceName () const ;

Returns a copy of the **source name**.

void setSourceName (const char * *name* = 0);

Sets the **source name**.

Object data (protected)

MQRMH *omqrmh*

The MQRMH data structure.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR
- MQRC_STRUC_LENGTH_ERROR
- MQRC_STRUC_ID_ERROR
- MQRC_INSUFFICIENT_DATA
- MQRC_INCONSISTENT_FORMAT
- MQRC_ENCODING_ERROR

ImqString C++ class

This class provides character string storage and manipulation for null-terminated strings.

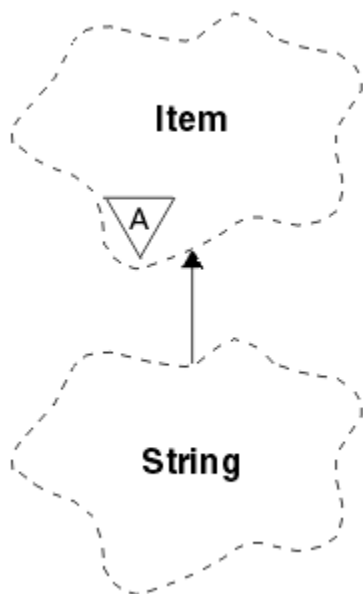


Figure 67. ImqString class

Use an ImqString in place of a **char *** in most situations where a parameter calls for a **char ***.

- [“Object attributes” on page 1918](#)
- [“Constructors” on page 1919](#)
- [“Class methods \(public\)” on page 1919](#)
- [“Overloaded ImqItem methods” on page 1919](#)
- [“Object methods \(public\)” on page 1919](#)
- [“Object methods \(protected\)” on page 1922](#)
- [“Reason codes” on page 1922](#)

Object attributes

characters

Characters in the **storage** that precede a trailing null.

length

Number of bytes in the **characters**. If there is no **storage**, the **length** is zero. The initial value is zero.

storage

A volatile array of bytes of arbitrary size. A trailing null must always be present in the **storage** after the **characters**, so that the end of the **characters** can be detected. Methods ensure that this situation

is maintained, but ensure, when setting bytes in the array directly, that a trailing null exists after modification. Initially, there is no **storage** attribute.

Constructors

ImqString();

The default constructor.

ImqString(const ImqString & *string*);

The copy constructor.

ImqString(const char *c*);

The **characters** comprise *c*.

ImqString(const char * *text*);

The **characters** are copied from *text*.

ImqString(const void * *buffer*, const size_t *length*);

Copies *length* bytes starting from *buffer* and assigns them to the **characters**. Substitution is made for any null characters copied. The substitution character is a period (.). No special consideration is given to any other non-printable or non-displayable characters copied.

Class methods (public)

static ImqBoolean copy(char * *destination-buffer*, const size_t *length*, const char * *source-buffer*, const char *pad* = 0);

Copies up to *length* bytes from *source-buffer* to *destination-buffer*. If the number of characters in *source-buffer* is insufficient, fills the remaining space in *destination-buffer* with *pad* characters. *source-buffer* can be zero. *destination-buffer* can be zero if *length* is also zero. Any error codes are lost. This method returns TRUE if successful.

static ImqBoolean copy (char * *destination-buffer*, const size_t *length*, const char * *source-buffer*, ImqError & *error-object*, const char *pad* = 0);

Copies up to *length* bytes from *source-buffer* to *destination-buffer*. If the number of characters in *source-buffer* is insufficient, fills the remaining space in *destination-buffer* with *pad* characters. *source-buffer* can be zero. *destination-buffer* can be zero if *length* is also zero. Any error codes are set in *error-object*. This method returns TRUE if successful.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & *msg*);

Copies the **characters** to the message buffer, replacing any existing content. Sets the *msg* **format** to MQFMT_STRING.

See the parent class method description for further details.

virtual ImqBoolean pasteIn (ImqMessage & *msg*);

Sets the **characters** by transferring the remaining data from the message buffer, replacing the existing **characters**.

To be successful, the **encoding** of the *msg* object must be MQENC_NATIVE. Retrieve messages with MQGMO_CONVERT to MQENC_NATIVE.

To be successful, the ImqMessage **format** must be MQFMT_STRING.

See the parent class method description for further details.

Object methods (public)

char & operator [] (const size_t *offset*) const ;

References the character at offset *offset* in the **storage**. Ensure that the relevant byte exists and is addressable.

ImqString operator () (const size_t *offset*, const size_t *length* = 1) const ;

Returns a substring by copying bytes from the **characters** starting at *offset*. If *length* is zero, returns the rest of the **characters**. If the combination of *offset* and *length* does not produce a reference within the **characters**, returns an empty ImqString.

void operator = (const ImqString & *string*);

Copies instance data from *string*, replacing the existing instance data.

ImqString operator + (const char *c*) const ;

Returns the result of appending *c* to the **characters**.

ImqString operator + (const char * *text*) const ;

Returns the result of appending *text* to the **characters**. This can also be inverted. For example:

```
strOne + "string two" ;  
"string one" + strTwo ;
```

Note: Although most compilers accept **strOne + "string two"**; Microsoft Visual C++ requires **strOne + (char *)"string two"**;

ImqString operator + (const ImqString & *string1*) const ;

Returns the result of appending *string1* to the **characters**.

ImqString operator + (const double *number*) const ;

Returns the result of appending *number* to the **characters** after conversion to text.

ImqString operator + (const long *number*) const ;

Returns the result of appending *number* to the **characters** after conversion to text.

void operator += (const char *c*);

Appends *c* to the **characters**.

void operator += (const char * *text*);

Appends *text* to the **characters**.

void operator += (const ImqString & *string*);

Appends *string* to the **characters**.

void operator += (const double *number*);

Appends *number* to the **characters** after conversion to text.

void operator += (const long *number*);

Appends *number* to the **characters** after conversion to text.

operator char * () const ;

Returns the address of the first byte in the **storage**. This value can be zero, and is volatile. Use this method only for read-only purposes.

ImqBoolean operator < (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. The result is TRUE if less than and FALSE if greater than or equal to.

ImqBoolean operator > (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. The result is TRUE if greater than and FALSE if less than or equal to.

ImqBoolean operator <= (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. The result is TRUE if less than or equal to and FALSE if greater than.

ImqBoolean operator >= (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. The result is TRUE if greater than or equal to and FALSE if less than.

ImqBoolean operator == (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. It returns either TRUE or FALSE.

ImqBoolean operator != (const ImqString & *string*) const ;

Compares the **characters** with those of *string* using the **compare** method. It returns either TRUE or FALSE.

short compare(const ImqString & *string*) const ;

Compares the **characters** with those of *string*. The result is zero if the **characters** are equal, negative if less than and positive if greater than. Comparison is case sensitive. A null ImqString is regarded as less than a nonnull ImqString.

ImqBoolean copyOut(char * *buffer*, const size_t *length*, const char *pad* = 0);

Copies up to *length* bytes from the **characters** to the *buffer*. If the number of **characters** is insufficient, fills the remaining space in *buffer* with *pad* characters. *buffer* can be zero if *length* is also zero. It returns TRUE if successful.

size_t copyOut(long & *number*) const ;

Sets *number* from the **characters** after conversion from text, and returns the number of characters involved in the conversion. If this is zero, no conversion has been performed and *number* is not set. A convertible character sequence must begin with the following values:

```
<blank(s)>
<+|->
digit(s)
```

size_t copyOut(ImqString & *token*, const char *c* = ' ') const ;

If the **characters** contain one or more characters that are different from *c*, identifies a token as the first contiguous sequence of such characters. In this case *token* is set to that sequence, and the value returned is the sum of the number of leading characters *c* and the number of bytes in the sequence. Otherwise, returns zero and does not set *token*.

size_t cutOut(long & *number*);

Sets *number* as for the **copy** method, but also removes from **characters** the number of bytes indicated by the return value. For example, the string shown in the following example can be cut into three numbers by using **cutOut** (*number*) three times:

```
strNumbers = "-1 0    +55 "
```

```
while ( strNumbers.cutOut( number ) );
```

number becomes -1, then 0, then 55
leaving *strNumbers* == " "

size_t cutOut(ImqString & *token*, const char *c* = ' ')

Sets *token* as for the **copyOut** method, and removes from **characters** the *strToken* characters and also any characters *c* that precede the *token* characters. If *c* is not a blank, removes characters *c* that directly succeed the *token* characters. Returns the number of characters removed. For example, the string shown in the following example can be cut into three tokens by using **cutOut** (*token*) three times:

```
strText = " Program Version 1.1 "
```

```
while ( strText.cutOut( token ) );
```

// *token* becomes "Program", then "Version",
// then "1.1" leaving *strText* == " "

The following example shows how to parse a DOS path name:

```
strPath = "C:\OS2\BITMAP\OS2LOGO.BMP"
```

```
strPath.cutOut( strDrive, ':' );
```

```
strPath.stripLeading( ':' );
```

```
while ( strPath.cutOut( strFile, '\\' ) );
```

// *strDrive* becomes "C".
// *strFile* becomes "OS2", then "BITMAP",
// then "OS2LOGO.BMP" leaving *strPath* empty.

ImqBoolean find(const ImqString & string);

Searches for an exact match for *string* anywhere within the **characters**. If no match is found, it returns FALSE. Otherwise, it returns TRUE. If *string* is null, it returns TRUE.

ImqBoolean find(const ImqString & string, size_t & offset);

Searches for an exact match for *string* somewhere within the **characters** from offset *offset* onwards. If *string* is null, it returns TRUE without updating *offset*. If no match is found, it returns FALSE (the value of *offset* might have been increased). If a match is found, it returns TRUE and updates *offset* to the offset of *string* within the **characters**.

size_t length() const ;

Returns the **length**.

ImqBoolean pasteIn(const double number, const char * format = "%f");

Appends *number* to the **characters** after conversion to text. It returns TRUE if successful.

The specification *format* is used to format the floating point conversion. If specified, it must be one suitable for use with **printf** and floating point numbers, for example **%3f**.

ImqBoolean pasteIn(const long number);

Appends *number* to the **characters** after conversion to text. It returns TRUE if successful.

ImqBoolean pasteIn(const void * buffer, const size_t length);

Appends *length* bytes from *buffer* to the **characters**, and adds a final trailing null. Substitutes any null characters copied. The substitution character is a period (.). No special consideration is given to any other nonprintable or nondisplayable characters copied. This method returns TRUE if successful.

ImqBoolean set(const char * buffer, const size_t length);

Sets the **characters** from a fixed-length character field, which might contain a null. Appends a null to the characters from the fixed-length field if necessary. This method returns TRUE if successful.

ImqBoolean setStorage(const size_t length);

Allocates (or reallocates) the **storage**. Preserves any original **characters**, including any trailing null, if there is still room for them, but does not initialize any additional storage.

This method returns TRUE if successful.

size_t storage() const ;

Returns the number of bytes in the **storage**.

size_t stripLeading(const char c = ' ');

Strips leading characters *c* from the **characters** and returns the number removed.

size_t stripTrailing(const char c = ' ');

Strips trailing characters *c* from the **characters** and returns the number removed.

ImqString upperCase() const ;

Returns an uppercase copy of the **characters**.

Object methods (protected)**ImqBoolean assign (const ImqString & string);**

Equivalent to the equivalent **operator =** method, but non-virtual. It returns TRUE if successful.

Reason codes

- MQRC_DATA_TRUNCATED
- MQRC_NULL_POINTER
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_BUFFER_ERROR
- MQRC_INCONSISTENT_FORMAT

ImqTrigger C++ class

This class encapsulates the MQTM (trigger message) data structure.

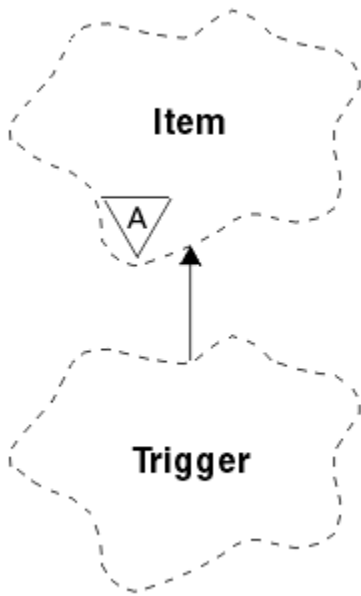


Figure 68. ImqTrigger class

Objects of this class are typically used by a trigger monitor program. The task of a trigger monitor program is to wait for these particular messages and act on them to ensure that other IBM MQ applications are started when messages are waiting for them.

See the IMQSTRG sample program for a usage example.

- [“Object attributes” on page 1923](#)
- [“Constructors” on page 1924](#)
- [“Overloaded ImqItem methods” on page 1924](#)
- [“Object methods \(public\)” on page 1924](#)
- [“Object data \(protected\)” on page 1925](#)
- [“Reason codes” on page 1925](#)

Object attributes

application id

Identity of the application that sent the message. The initial value is a null string.

application type

Type of application that sent the message. The initial value is zero. The following additional values are possible:

- MQAT_AIX
- MQAT_CICS
- MQAT_DOS
- MQAT_IMS
- MQAT_MVS
- MQAT_NOTES_AGENT
- MQAT_OS2
- MQAT_OS390
- MQAT_OS400

- MQAT_UNIX
- MQAT_WINDOWS
- MQAT_WINDOWS_NT
- MQAT_USER_FIRST
- MQAT_USER_LAST

environment data

Environment data for the process. The initial value is a null string.

process name

Process name. The initial value is a null string.

queue name

Name of the queue to be started. The initial value is a null string.

trigger data

Trigger data for the process. The initial value is a null string.

user data

User data for the process. The initial value is a null string.

Constructors

ImqTrigger();

The default constructor.

ImqTrigger(const ImqTrigger & trigger);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut (ImqMessage & msg);

Writes an MQTM data structure to the message buffer, replacing any existing content. Sets the *msg format* to MQFMT_TRIGGER.

See the ImqItem class method description at [“ImqItem C++ class” on page 1868](#) for further details.

virtual ImqBoolean pasteIn (ImqMessage & msg);

Reads an MQTM data structure from the message buffer.

To be successful, the ImqMessage **format** must be MQFMT_TRIGGER.

See the ImqItem class method description at [“ImqItem C++ class” on page 1868](#) for further details.

Object methods (public)

void operator = (const ImqTrigger & trigger);

Copies instance data from *trigger*, replacing the existing instance data.

ImqString applicationId () const ;

Returns a copy of the **application id**.

void setApplicationId (const char * id);

Sets the **application id**.

MQLONG applicationType () const ;

Returns the **application type**.

void setApplicationType (const MQLONG type);

Sets the **application type**.

ImqBoolean copyOut (MQTMC2 * ptmc2);

Encapsulates the MQTM data structure, which is the one received on initiation queues. Fills in an equivalent MQTMC2 data structure provided by the caller, and sets the QMgrName field (which is not present in the MQTM data structure) to all blanks. The MQTMC2 data structure is traditionally used as a parameter to applications started by a trigger monitor. This method returns TRUE if successful.

ImqString environmentData () const ;

Returns a copy of the **environment data**.

void setEnvironmentData (const char * data);

Sets the **environment data**.

ImqString processName () const ;

Returns a copy of the **process name**.

void setProcessName (const char * name);

Sets the **process name**, padded with blanks to 48 characters.

ImqString queueName () const ;

Returns a copy of the **queue name**.

void setQueueName (const char * name);

Sets the **queue name**, padding with blanks to 48 characters.

ImqString triggerData () const ;

Returns a copy of the **trigger data**.

void setTriggerData (const char * data);

Sets the **trigger data**.

ImqString userData () const ;

Returns a copy of the **user data**.

void setUserData (const char * data);

Sets the **user data**.

Object data (protected)

MQTM omqtm

The MQTM data structure.

Reason codes

- MQRC_NULL_POINTER
- MQRC_INCONSISTENT_FORMAT
- MQRC_ENCODING_ERROR
- MQRC_STRUC_ID_ERROR

ImqWorkHeader C++ class

This class encapsulates specific features of the MQWIH data structure.

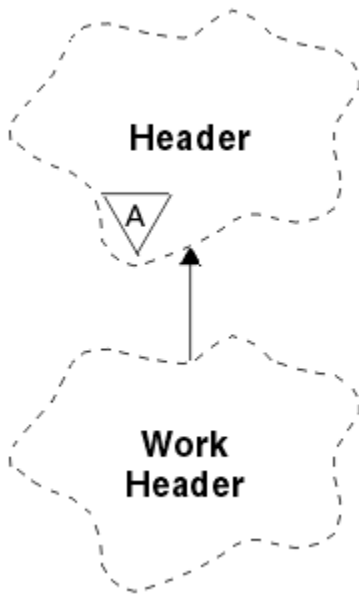


Figure 69. *ImqWorkHeader* class

Objects of this class are used by applications putting messages to the queue managed by the z/OS Workload Manager.

- [“Object attributes” on page 1926](#)
- [“Constructors” on page 1926](#)
- [“Overloaded ImqItem methods” on page 1927](#)
- [“Object methods \(public\)” on page 1927](#)
- [“Object data \(protected\)” on page 1927](#)
- [“Reason codes” on page 1927](#)

Object attributes

message token

Message token for the z/OS Workload Manager, of length MQ_MSG_TOKEN_LENGTH. The initial value is MQMTOK_NONE.

service name

The 32-character name of a process. The name is initially blanks.

service step

The 8-character name of a step within the process. The name is initially blanks.

Constructors

ImqWorkHeader();

The default constructor.

ImqWorkHeader(const ImqWorkHeader & header);

The copy constructor.

Overloaded ImqItem methods

virtual ImqBoolean copyOut(ImqMessage & msg);

Inserts an MQWIH data structure into the beginning of the message buffer, moving the existing message data further along, and sets the *msg format* to MQFMT_WORK_INFO_HEADER.

See the parent class method description for more details.

virtual ImqBoolean pasteIn(ImqMessage & msg);

Reads an MQWIH data structure from the message buffer.

To be successful, the encoding of the *msg* object must be MQENC_NATIVE. Retrieve messages with MQGMO_CONVERT to MQENC_NATIVE.

The ImqMessage format must be MQFMT_WORK_INFO_HEADER.

See the parent class method description for more details.

Object methods (public)

void operator = (const ImqWorkHeader & header);

Copies instance data from *header*, replacing the existing instance data.

ImqBinary messageToken () const;

Returns the **message token**.

ImqBoolean setMessageToken(const ImqBinary & token);

Sets the **message token**. The data length of *token* must be either zero or MQ_MSG_TOKEN_LENGTH. It returns TRUE if successful.

void setMessageToken(const MQBYTE16 token = 0);

Sets the **message token**. *token* can be zero, which is the same as specifying MQMTOK_NONE. If *token* is nonzero, it must address MQ_MSG_TOKEN_LENGTH bytes of binary data.

When using predefined values such as MQMTOK_NONE, you might need make a cast to ensure a signature match; for example, (MQBYTE *)MQMTOK_NONE.

ImqString serviceName () const;

Returns the **service name**, including trailing blanks.

void setServiceName(const char * name);

Sets the **service name**.

ImqString serviceStep () const;

Returns the **service step**, including trailing blanks.

void setServiceStep(const char * step);

Sets the **service step**.

Object data (protected)

MQWIH omqwih

The MQWIH data structure.

Reason codes

- MQRC_BINARY_DATA_LENGTH_ERROR

Properties of IBM MQ classes for JMS objects

All objects in IBM MQ classes for JMS have properties. Different properties apply to different object types. Different properties have different allowable values, and symbolic property values differ between the administration tool and program code.

IBM MQ classes for JMS provides facilities to set and query the properties of objects using the IBM MQ JMS administration tool, IBM MQ Explorer, or in an application. Many of the properties are relevant only to a specific subset of the object types.

For information on how you use the IBM MQ JMS administration tool, see [Configuring JMS objects using the administration tool](#).

Table 677 on page 1928 gives a brief description of each property and shows for each property which object types it applies to. The object types are identified using keywords; see [Configuring JMS objects using the administration tool](#) for an explanation of these objects.

Numbers refer to notes at the end of the table. See also “Dependencies between properties of IBM MQ classes for JMS objects” on page 1931.

A property consists of a name-value pair in the format:

```
PROPERTY_NAME(property_value)
```

The topics in this section list, for each property, the name of the property and a brief description, and shows the valid property values used in the administration tool. and the set method that is used to set the value of the property in an application. The topics also show the valid property values for each property and the mapping between symbolic property values used in the tool and their programmable equivalents.

Property names are not case-sensitive, and are restricted to the set of recognized names shown in these topics.

Table 677. Property names and applicable object types									
Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
“APPLICATIONNAME” on page 1933	APPNAME	Y	Y	Y			Y	Y	Y
“ASYNCEXCEPTION” on page 1933	AEX	Y	Y	Y			Y	Y	Y
“BROKERCCDURSUBQ” on page 1934 ¹	CCDSUB					Y			
“BROKERCCSUBQ” on page 1935 ¹	CCSUB	Y		Y			Y		Y
“BROKERCONQ” on page 1935 ¹	BCON	Y		Y			Y		Y
“BROKERDURSUBQ” on page 1936 ¹	BDSUB					Y			
“BROKERPUBQ” on page 1936 ¹	BPUB	Y		Y		Y	Y		Y
“BROKERPUBQMGR” on page 1937 ¹	BPQM					Y			
“BROKERQMGR” on page 1937 ¹	BQM	Y		Y			Y		Y
“BROKERSUBQ” on page 1937 ¹	BSUB	Y		Y			Y		Y
“BROKERVER” on page 1938 ¹	BVER	Y ²		Y ²		Y	Y		Y
“CCDTURL” on page 1939 ³	CCDT	Y	Y	Y			Y	Y	Y
“CCSID” on page 1939	CCS	Y	Y	Y	Y	Y	Y	Y	Y
“CHANNEL” on page 1940 ³	CHAN	Y	Y	Y			Y	Y	Y
“CLEANUP” on page 1940 ¹	CL	Y		Y			Y		Y

Table 677. Property names and applicable object types (continued)

Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
“CLEANUPINT” on page 1941 ¹	CLINT	Y		Y			Y		Y
“CONNECTIONNAMELIST” on page 1941	CNLIST	Y	Y	Y					
“CLIENTRECONNECTOPTIONS” on page 1941	CROPT	Y	Y	Y					
“CLIENTRECONNECTTIMEOUT” on page 1942	CRT	Y	Y	Y					
“CLIENTID” on page 1943	CID	Y ²	Y	Y ²			Y	Y	Y
“CLONESUPP” on page 1943	CLS	Y		Y			Y		Y
“COMPHDR” on page 1944	HC	Y		Y			Y		Y
“COMPMSG” on page 1944	MC	Y	Y	Y			Y	Y	Y
“CONNOPT” on page 1945	CNOPT	Y	Y	Y			Y	Y	Y
“CONNTAG” on page 1946	CNTAG	Y	Y	Y			Y	Y	Y
“DESCRIPTION” on page 1946	DESC	Y ²	Y	Y ²	Y	Y	Y	Y	Y
“DIRECTAUTH” on page 1947	DAUTH	Y ²		Y ²					
“ENCODING” on page 1947	ENC				Y	Y			
“EXPIRY” on page 1948	EXP				Y	Y			
“FAILIFQUIESCE” on page 1948	FIQ	Y	Y	Y	Y	Y	Y	Y	Y
“HOSTNAME” on page 1949	HOST	Y ²	Y	Y ²			Y	Y	Y
“LOCALADDRESS” on page 1950	LA	Y ²	Y	Y ²			Y	Y	Y
“MAPNAMESTYLE” on page 1950	MNST	Y	Y	Y			Y	Y	Y
“MAXBUFFSIZE” on page 1951	MBSZ	Y ²		Y ²					
“MDREAD” on page 1951	MDR				Y	Y			
“MDWRITE” on page 1952	MDW				Y	Y			
“MDMSGCTX” on page 1952	MDCTX				Y	Y			
“MSGBATCHSZ” on page 1953 ¹	MBS	Y	Y	Y			Y	Y	Y
“MSGBODY” on page 1953	MBODY				Y	Y			
“MSGRETENTION” on page 1954	MRET	Y	Y				Y	Y	
“MSGSELECTION” on page 1954 ¹	MSEL	Y		Y			Y		Y
“MULTICAST” on page 1955	MCAST	Y ²		Y ²		Y			
“OPTIMISTICPUBLICATION” on page 1956 ¹	OPTPUB	Y		Y					
“OUTCOMENOTIFICATION” on page 1956 ¹	NOTIFY	Y		Y					
“PERSISTENCE” on page 1957	PER				Y	Y			

Table 677. Property names and applicable object types (continued)

Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
“POLLINGINT” on page 1957 ¹	PINT	Y	Y	Y			Y	Y	Y
“PORT” on page 1958	PORT	Y ²	Y	Y ²			Y	Y	Y
“PRIORITY” on page 1958	PRI				Y	Y			
“PROCESSDURATION” on page 1959 ¹	PROC DUR	Y		Y					
“PROVIDERVERSION” on page 1959	PVER	Y	Y	Y			Y	Y	Y
“PROXYHOSTNAME” on page 1962	PHOST	Y ²		Y ²					
“PROXYPORT” on page 1962	PPORT	Y ²		Y ²					
“PUBACKINT” on page 1963 ¹	PAI	Y		Y			Y		Y
“PUTASYNCAALLOWED” on page 1963	PAALD				Y	Y			
“QMANAGER” on page 1964	QMGR	Y	Y	Y	Y		Y	Y	Y
“QUEUE” on page 1964	QU				Y				
“READAHEADALLOWED” on page 1965	RAALD				Y	Y			
“READAHEADCLOSEPOLICY” on page 1965	RACP				Y	Y			
“RECEIVECCSID” on page 1966	RCCS				Y	Y			
“RECEIVECONVERSION” on page 1966	RCNV				Y	Y			
“RECEIVEISOLATION” on page 1967 ¹	RCVISOL	Y		Y					
“RECEXIT” on page 1967	RCX	Y	Y	Y			Y	Y	Y
“RECEXITINIT” on page 1968	RCXI	Y	Y	Y			Y	Y	Y
“REPLYTOSTYLE” on page 1968	RTOST				Y	Y			
“RESCANINT” on page 1969 ¹	RINT	Y	Y				Y	Y	
“SECEXIT” on page 1969	SCX	Y	Y	Y			Y	Y	Y
“SECEXITINIT” on page 1970	SCXI	Y	Y	Y			Y	Y	Y
“SENDCHECKCOUNT” on page 1970	SCC	Y	Y	Y			Y	Y	Y
“SENDEXIT” on page 1971	SDX	Y	Y	Y			Y	Y	Y
“SENDEXITINIT” on page 1971	SDXI	Y	Y	Y			Y	Y	Y
“SHARECONVALLOWED” on page 1972	SCALD	Y	Y	Y			Y	Y	Y
“SPARSESUBS” on page 1972 ¹	SSUBS	Y		Y					
“SSLCIPHERSUITE” on page 1973	SCPHS	Y	Y	Y			Y	Y	Y
“SSLCRL” on page 1973	SCRL	Y	Y	Y			Y	Y	Y
“SSLFIPSREQUIRED” on page 1974	SFIPS	Y	Y	Y			Y	Y	Y

Table 677. Property names and applicable object types (continued)

Property	Short form	Object type							
		CF	QCF	TCF	Q	T	XACF	XAQCF	XATCF
“SSLPEERNAME” on page 1974	SPEER	Y	Y	Y			Y	Y	Y
“SSLRESETCOUNT” on page 1975	SRC	Y	Y	Y			Y	Y	Y
“STATREFRESHINT” on page 1975 ¹	SRI	Y		Y			Y		Y
“SUBSTORE” on page 1976 ¹	SS	Y		Y			Y		Y
“SYNCPOINTALLGETS” on page 1976	SPAG	Y	Y	Y			Y	Y	Y
“TARGCLIENT” on page 1977	TC				Y	Y			
“TARGCLIENTMATCHING” on page 1977	TCM	Y	Y				Y	Y	
“TEMPMODEL” on page 1978	TM	Y	Y				Y	Y	
“TEMPQPREFIX” on page 1978	TQP	Y	Y				Y	Y	
“TEMPTOPICPREFIX” on page 1979	TTP	Y		Y			Y		Y
“TOPIC” on page 1979	TOP					Y			
“TRANSPORT” on page 1979	TRAN	Y ²	Y	Y ²			Y	Y	Y
“WILDCARDFORMAT” on page 1980	WCFMT	Y		Y			Y		Y

Note:

1. This property can be used with Version 7.0 of IBM MQ classes for JMS but has no effect for an application connected to a Version 7.0 queue manager unless the PROVIDERVERSION property of the connection factory is set to a version number less than 7.
2. Only the BROKERVER, CLIENTID, DESCRIPTION, DIRECTAUTH, HOSTNAME, LOCALADDRESS, MAXBUFFSIZE, MULTICAST, PORT, PROXYHOSTNAME, PROXYPORT, and TRANSPORT properties are supported for a ConnectionFactory or TopicConnectionFactory object when using a real-time connection to a broker.
3. The CCDTURL and CHANNEL properties of an object must not both be set at the same time.

Dependencies between properties of IBM MQ classes for JMS objects

The validity of some properties is dependent on the particular values of other properties.

This dependency can occur in the following groups of properties:

- Client properties
- Properties for a real-time connection to a broker
- Exit initialization strings

Client properties

For a connection to a queue manager, the following properties are relevant only if TRANSPORT has the value CLIENT:

- HOSTNAME
- PORT
- CHANNEL
- LOCALADDRESS

- CCDTURL
- CCSID
- COMPHDR
- COMPMSG
- RECEXIT
- RECEXITINIT
- SECEXIT
- SECEXITINIT
- SENDEXIT
- SENDEXITINIT
- SHARECONVALLOWED
- SSLCIPHERSUITE
- SSLCRL
- SSLFIPSREQUIRED
- SSLPEERNAME
- SSLRESETCOUNT
- APPLICATIONNAME

You cannot set values for these properties by using the administration tool if TRANSPORT has the value BIND.

If TRANSPORT has the value CLIENT, the default value of the BROKERVER property is V1 and the default value of the PORT property is 1414. If you set the value of BROKERVER or PORT explicitly, a later change to the value of TRANSPORT does not override your choices.

Properties for a real-time connection to a broker

Only the following properties are relevant if TRANSPORT has the value DIRECT or DIRECTHTTP:

- BROKERVER
- CLIENTID
- DESCRIPTION
- DIRECTAUTH
- HOSTNAME
- LOCALADDRESS
- MAXBUFFSIZE
- MULTICAST (supported only for DIRECT)
- PORT
- PROXYHOSTNAME (supported only for DIRECT)
- PROXYPORT (supported only for DIRECT)

If TRANSPORT has the value DIRECT or DIRECTHTTP, the default value of the BROKERVER property is V2, and the default value of the PORT property is 1506. If you set the value of BROKERVER or PORT explicitly, a later change to the value of TRANSPORT does not override your choices.

Exit initialization strings

Do not set any of the exit initialization strings without supplying the corresponding exit name. The exit initialization properties are:

- RECEXITINIT
- SECEXITINIT
- SENDEXITINIT

For example, specifying `RECEXITINIT(myString)` without specifying `RECEXIT(some.exit.classname)` causes an error.

Related reference

[“TRANSPORT” on page 1979](#)

The nature of a connection to a queue manager or broker.

APPLICATIONNAME

An application can set a name that identifies its connection to the queue manager. This application name is shown by the **DISPLAY CONN MQSC/PCF** command (where the field is called **APPLTAG**) or in the IBM MQ Explorer **Application Connections** display (where the field is called **App name**).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: APPLICATIONNAME

JMS administration tool short name: APPNAME

Programmatic access

Setters/getters

- `MQConnectionFactory.setAppName()`
- `MQConnectionFactory.getAppName()`

Values

Any valid string that is no longer than 28 characters. Longer names are adjusted to fit by removing leading package names, if necessary. For example, if the invoking class is `com.example.MainApp`, the full name is used, but if the invoking class is `com.example.dictionaryAndThesaurus.multilingual.mainApp`, the name `multilingual.mainApp` is used, because it is the longest combination of class name and rightmost package name that fits into the available length.

If the class name itself is more than 28 characters long, it is truncated to fit. For example, `com.example.mainApplicationForSecondTestCase` becomes `mainApplicationForSecondTest`.

ASYNCEXCEPTION

This property determines whether IBM MQ classes for JMS informs an `ExceptionListener` only when a connection is broken, or when any exception occurs asynchronously to a JMS API call. This applies to all Connections created from this `ConnectionFactory` that have an `ExceptionListener` registered.

Applicable objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: ASYNCEXCEPTION

JMS administration tool short name: AEX

Programmatic access

Setters/Getters

- `MQConnectionFactory.setAsyncExceptions()`

- `MQConnectionFactory.getAsyncExceptions()`

Values

ASYNC_EXCEPTIONS_ALL

Any exception detected asynchronously, outside the scope of a synchronous API call, and all connection broken exceptions are sent to the `ExceptionListener`.

Environment	Value
JMS Administration Tool	ALL
Programmatic	<code>WMQCONSTANTS.ASYNC_EXCEPTIONS_ALL = -1</code>
IBM MQ Explorer	All

ASYNC_EXCEPTIONS_CONNECTIONBROKEN

Only exceptions indicating a broken connection are sent to the `ExceptionListener`. Any other exceptions occurring during asynchronous processing are not reported to the `ExceptionListener`, and hence the application is not informed of these exceptions. **V 8.0.0.2** This is the default value from IBM MQ 8.0.0, Fix Pack 2 (see [JMS: Exception listener changes in Version 8.0](#)).

Environment	Value
JMS Administration Tool	CONNECTIONBROKEN
Programmatic	<code>WMQCONSTANTS.ASYNC_EXCEPTIONS_CONNECTIONBROKEN = 1</code>
MQ Explorer	Connection Broken

The following additional constant is defined:

- **V 8.0.0.2** From Version 8.0.0, Fix Pack 2: `WMQCONSTANTS.ASYNC_EXCEPTIONS_DEFAULT = ASYNC_EXCEPTIONS_CONNECTIONBROKEN`
- Before Version 8.0.0, Fix Pack 2: `WMQCONSTANTS.ASYNC_EXCEPTIONS_DEFAULT = ASYNC_EXCEPTIONS_ALL`

Related concepts

[Exceptions in IBM MQ classes for JMS](#)

BROKERCCDURSUBQ

The name of the queue from which durable subscription messages are retrieved for a `ConnectionConsumer`.

Applicable objects

Topic

JMS administration tool long name: `BROKERCCDURSUBQ`

JMS administration tool short name: `CCDSUB`

Programmatic access

Setters/getters

- `MQTopic.setBrokerCCDurSubQueue()`
- `MQTopic.getBrokerCCDurSubQueue()`

Values

SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE

This is the default value.

Any valid string

BROKERCCSUBQ

The name of the queue from which non-durable subscription messages are retrieved for a ConnectionConsumer.

Applicable objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: BROKERCCSUBQ

JMS administration tool short name: CCSUB

Programmatic access

Setters/getters

- MQConnectionFactory.setBrokerCCSubQueue()
- MQConnectionFactory.getBrokerCCSubQueue()

Values

SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE

This is the default value.

Any valid string

BROKERCONQ

The control queue name of the broker.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: BROKERCONQ

JMS administration tool short name: BCON

Programmatic access

Setters/getters

- MQConnectionFactory.setBrokerControlQueue()
- MQConnectionFactory.getBrokerControlQueue()

Values

SYSTEM.BROKER.CONTROL.QUEUE

This is the default value.

Any valid string

BROKERDURSUBQ

When the IBM MQ classes for JMS are being used in IBM MQ messaging provider migration mode, this property specifies the name of the queue from which durable subscription messages are retrieved.

Applicable objects

Topic

JMS administration tool long name: BROKERDURSUBQ

JMS administration tool short name: BDSUB

Programmatic access

Setters/getters

- `MQTopic.setBrokerDurSubQueue()`
- `MQTopic.getBrokerDurSubQueue()`

Values

SYSTEM.JMS.D.SUBSCRIBER.QUEUE

This is the default value.

Any valid string

Starting with `SYSTEM.JMS.D`

Related concepts

[Rules for selecting the IBM MQ messaging provider mode](#)

BROKERPUBQ

The name of the queue where published messages are sent (the stream queue).

Applicable Objects

ConnectionFactory, TopicConnectionFactory, Topic, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: BROKERPUBQ

JMS administration tool short name: BPUB

Programmatic access

Setters/getters

- `MQConnectionFactory.setBrokerPubQueue`
- `MQConnectionFactory.getBrokerPubQueue`

Values

SYSTEM.BROKER.DEFAULT.STREAM

This is the default value.

Any valid string

BROKERPUBQMGR

The name of the queue manager that owns the queue where messages published on the topic are sent.

Applicable Objects

Topic

JMS administration tool long name: BROKERPUBQMGR

JMS administration tool short name: BPQM

Programmatic access

Setters/getters

- `MQTopic.setBrokerPubQueueManager()`
- `MQTopic.getBrokerPubQueueManager()`

Values

null

This is the default value.

Any valid string

BROKERQMGR

The name of the queue manager on which the broker is running.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: BROKERQMGR

JMS administration tool short name: BQM

Programmatic access

Setters/getters

- `MQConnectionFactory.setBrokerQueueManager()`
- `MQConnectionFactory.getBrokerQueueManager()`

Values

null

This is the default value.

Any valid string

BROKERSUBQ

When the IBM MQ classes for JMS are being used in IBM MQ messaging provider migration mode, this property specifies the name of the queue from which non-durable subscription messages are retrieved.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: BROKERSUBQ

JMS administration tool short name: BSUB

Programmatic access

Setters/getters

- `MQConnectionFactory.setBrokerSubQueue()`
- `MQConnectionFactory.getBrokerSubQueue()`

Values

SYSTEM.JMS.ND.SUBSCRIBER.QUEUE

This is the default value.

Any valid string

Starting with `SYSTEM.JMS.ND`

Related concepts

[Rules for selecting the IBM MQ messaging provider mode](#)

BROKERVER

The version of the broker being used.

Applicable Objects

`ConnectionFactory`, `TopicConnectionFactory`, `Topic`, `XAConnectionFactory`, `XATopicConnectionFactory`

JMS administration tool long name: BROKERVER

JMS administration tool short name: BVER

Programmatic access

Setters/getters

- `MQConnectionFactory.setBrokerVersion()`
- `MQConnectionFactory.getBrokerVersion()`

Values

V1

To use an IBM MQ Publish/Subscribe broker, or to use a broker of IBM MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in compatibility mode. This is the default value if `TRANSPORT` is set to `BIND` or `CLIENT`.

V2

To use a broker of IBM MQ Integrator, WebSphere Event Broker, WebSphere Business Integration Event Broker, or WebSphere Business Integration Message Broker in native mode. This is the default value if `TRANSPORT` is set to `DIRECT` or `DIRECTHTTP`.

unspecified

After the broker has migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant.

CCDTURL

A Uniform Resource Locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CCDTURL

JMS administration tool short name: CCDT

Programmatic access

Setters/getters

- MQConnectionFactory.setCCDTURL()
- MQConnectionFactory.getCCDTURL()

Values

null

This is the default value.

A Uniform Resource Locator (URL)

CCSID

The coded character set ID to be used for a connection or destination.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, Queue, Topic, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CCSID

JMS administration tool short name: CCS

Programmatic access

Setters/getters

- MQConnectionFactory.setCCSID()
- MQConnectionFactory.getCCSID()

Values

819

This is the default value for a connection factory.

1208

This is the default value for a destination.

Any positive integer

CHANNEL

The name of the client connection channel being used.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CHANNEL

JMS administration tool short name: CHAN

Programmatic access

Setters/getters

- MQConnectionFactory.setChannel()
- MQConnectionFactory.getChannel()

Values

SYSTEM.DEF.SVRCONN

This is the default value.

Any valid string

CLEANUP

Cleanup Level for BROKER or MIGRATE Subscription Stores.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CLEANUP

JMS administration tool short name: CL

Programmatic access

Setters/getters

- MQConnectionFactory.setCleanupLevel()
- MQConnectionFactory.getCleanupLevel()

Values

SAFE

Use safe cleanup. This is the default value.

ASPROP

Use safe, strong, or no cleanup according to a property set on the Java command line.

NONE

Use no cleanup.

STRONG

Use strong cleanup.

CLEANUPINT

The interval, in milliseconds, between background executions of the publish/subscribe cleanup utility.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CLEANUPINT

JMS administration tool short name: CLINT

Programmatic access

Setters/getters

- MQConnectionFactory.setCleanupInterval()
- MQConnectionFactory.getCleanupInterval()

Values

3600000

This is the default value.

Any positive integer

CONNECTIONNAMELIST

List of TCP/IP connection names. The list is tried in order, once per each reconnection retry attempt.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory

JMS administration tool long name: CONNECTIONNAMELIST

JMS administration tool short name: CNLIST

Programmatic access

Setters/getters

- MQConnectionFactory.setconnectionNameList()
- MQConnectionFactory.getconnectionNameList()

Values

Comma separated list of HOSTNAME(PORT). HOSTNAME can be either a DNS name or IP address.

PORT defaults to 1414.

CLIENTRECONNECTOPTIONS

Options governing reconnection.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory

JMS administration tool long name: CLIENTRECONNECTOPTIONS

JMS administration tool short name: CROPT

Programmatic access

Setters/getters

- `MQConnectionFactory.setClientReconnectOptions()`
- `MQConnectionFactory.getClientReconnectOptions()`

Values

QMGR

The application can reconnect, but only to the queue manager to which it originally connected.

Use this value if an application can be reconnected, but there is an affinity between the IBM MQ classes for JMS application, and the queue manager to which it first established a connection.

Choose this value if you want an application to automatically reconnect to the standby instance of a highly available queue manager.

To use this value programmatically, use the constant `WMQConstants.WMQ_CLIENT_RECONNECT_Q_MGR`.

ANY

The application can reconnect to any queue manager.

Use the reconnect option only if there is no affinity between the IBM MQ classes for JMS application and the queue manager with which it initially established a connection.

To use this value from a program, use the constant `WMQConstants.WMQ_CLIENT_RECONNECT`.

DISABLED

The application will not be reconnected.

To use this value programmatically, use the constant `WMQConstants.WMQ_CLIENT_RECONNECT_DISABLED`.

ASDEF

Whether the application will reconnect automatically depends on the value of the IBM MQ channel attribute `DefReconnect`.

To use this value from a program, use the constant `WMQConstants.WMQ_CLIENT_RECONNECT_AS_DEF`.

CLIENTRECONNECTTIMEOUT

Time before reconnection retries cease.

Applicable Objects

`ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`

JMS administration tool long name: `CLIENTRECONNECTTIMEOUT`

JMS administration tool short name: `CRT`

Programmatic access

Setters/getters

- `MQConnectionFactory.setClientReconnectTimeout()`
- `MQConnectionFactory.getClientReconnectTimeout()`

Values

Interval in seconds. Default 1800 (30 minutes).

CLIENTID

The client identifier is used to uniquely identify the application connection for durable subscriptions.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CLIENTID

JMS administration tool short name: CID

Programmatic access

Setters/getters

- MQConnectionFactory.setClientId()
- MQConnectionFactory.getClientId()

Values

null

This is the default value.

Any valid string

CLONESUPP

Whether two or more instances of the same durable topic subscriber can run simultaneously.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CLONESUPP

JMS administration tool short name: CLS

Programmatic access

Setters/getters

- MQConnectionFactory.setCloneSupport()
- MQConnectionFactory.getCloneSupport()

Values

DISABLED

Only one instance of a durable topic subscriber can run at a time. This is the default value.

ENABLED

Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).

COMPHDR

A list of the techniques that can be used for compressing header data on a connection.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: COMPHDR

JMS administration tool short name: HC

Programmatic access

Setters/getters

- MQConnectionFactory.setHdrCompList()
- MQConnectionFactory.getHdrCompList()

Values

NONE

This is the default value.

SYSTEM

RLE message header compression is performed.

COMPMSG

A list of the techniques that can be used for compressing message data on a connection.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: COMPMSG

JMS administration tool short name: MC

Programmatic access

Setters/getters

- MQConnectionFactory.setMsgCompList()
- MQConnectionFactory.getMsgCompList()

Values

NONE

This is the default value.

A list of one or more of the following values separated by blank characters:

RLE ZLIBFAST ZLIBHIGH

CONNOPT

Controls how IBM MQ classes for JMS applications that use the bindings transport connect to the queue manager.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory.

JMS administration tool long name: CONNOPT

JMS administration tool short name: CNOPT

Programmatic access

Setters/getters

- `MQConnectionFactory.setMQConnectionOptions()`
- `MQConnectionFactory.getMQConnectionOptions()`

Values

STANDARD

The nature of the binding between the application and the queue manager depends on the value of the *DefaultBindType* attribute of the queue manager. The STANDARD value maps to the IBM MQ *ConnectOption* MQCNO_STANDARD_BINDING.

SHARED

The application and the local queue manager agent run in separate units of execution but share some resources. This value maps to the IBM MQ *ConnectOption* MQCNO_SHARED_BINDING.

ISOLATED

The application and the local queue manager agent run in separate units of execution and share no resources. The ISOLATED value maps to the IBM MQ *ConnectOption* MQCNO_ISOLATED_BINDING.

FASTPATH

The application and the local queue manager agent run in the same unit of execution. This value maps to the IBM MQ *ConnectOption* MQCNO_FASTPATH_BINDING.

SERIALQM

The application requests exclusive use of the connection tag within the scope of the queue manager. This value maps to the IBM MQ *ConnectOption* MQCNO_SERIALIZE_CONN_TAG_Q_MGR.

SERIALQSG

The application requests exclusive use of the connection tag within the scope of the queue sharing group to which the queue manager belongs. The SERIALQSG value maps to the IBM MQ *ConnectOption* MQCNO_SERIALIZE_CONN_TAG_QSG.

RESTRICTQM

The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue manager. This value maps to the IBM MQ *ConnectOption* MQCNO_RESTRICT_CONN_TAG_Q_MGR.

RESTRICTQSG

The application requests shared use of the connection tag, but there are restrictions on the shared use of the connection tag within the scope of the queue sharing group to which the queue manager belongs. This value maps to the IBM MQ *ConnectOption* MQCNO_RESTRICT_CONN_TAG_QSG.

For further information on IBM MQ connection options, see [Connecting to a queue manager using the MQCONN call](#).

CONNTAG

A tag that the queue manager associates with the resources updated by the application within a unit of work while the application is connected to the queue manager.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: CONNTAG

JMS administration tool short name: CNTAG

Programmatic access

Setters/getters

- MQConnectionFactory.setConnTag()
- MQConnectionFactory.getConnTag()

Values

A byte array of 128 elements, where each element is 0

This is the default value.

Any string

The value is truncated if it is longer than 128 bytes.

DESCRIPTION

A description of the stored object.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, Queue, Topic, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: DESCRIPTION

JMS administration tool short name: DESC

Programmatic access

Setters/getters

- MQConnectionFactory.setDescription()
- MQConnectionFactory.getDescription()

Values

null

This is the default value.

Any valid string

DIRECTAUTH

Whether SSL authentication is used on a real-time connection to a broker.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: DIRECTAUTH

JMS administration tool short name: DAUTH

Programmatic access

Setters/getters

- MQConnectionFactory.setDirectAuth()
- MQConnectionFactory.getDirectAuth()

Values

BASIC

No authentication, username authentication, or password authentication. This is the default value.

CERTIFICATE

Public key certificate authentication.

ENCODING

How numeric data in the body of a message is represented when the message is sent to this destination. The property specifies the representation of binary integers, packed decimal integers, and floating point numbers.

Applicable Objects

Queue, Topic

JMS administration tool long name: ENCODING

JMS administration tool short name: ENC

Programmatic access

Setters/getters

- MQDestination.setEncoding()
- MQDestination.getEncoding()

Values

ENCODING property

The valid values that the ENCODING property can take are constructed from the three sub-properties:

integer encoding

Either normal or reversed

decimal encoding

Either normal or reversed

floating-point encoding

IEEE normal, IEEE reversed, or z/OS

The ENCODING property is expressed as a three-character string with the following syntax:

```
{N|R}{N|R}{N|R|3}
```

In this string:

- N denotes normal
- R denotes reversed
- 3 denotes z/OS
- The first character represents *integer encoding*
- The second character represents *decimal encoding*
- The third character represents *floating-point encoding*

This provides a set of twelve possible values for the ENCODING property.

There is an additional value, the string NATIVE, which sets appropriate encoding values for the Java platform.

The following examples show valid combinations for ENCODING:

```
ENCODING (NNR)  
ENCODING (NATIVE)  
ENCODING (RR3)
```

EXPIRY

The time after which messages at a destination expire.

Applicable Objects

Queue, Topic

JMS administration tool long name: EXPIRY

JMS administration tool short name: EXP

Programmatic access

Setters/getters

- MQDestination.setExpiry()
- MQDestination.getExpiry()

Values

APP

Expiry can be defined by the JMS application. This is the default value.

UNLIM

No expiry occurs.

0

No expiry occurs.

Any positive integer representing expiry in milliseconds.

FAILIFQUIESCE

This property determines whether calls to certain methods fail if either the queue manager is in a quiescing state, or an application is connecting to a queue manager using the CLIENT transport and the

channel that the application is using has been put into a quiescing state, for example, by using the **STOP CHANNEL** or **STOP CHANNEL MODE(QUIESCE)** MQSC command.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, Queue, Topic, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: FAILIFQUIESCE

JMS administration tool short name: FIQ

Programmatic access

Setters/getters

- MQConnectionFactory.setFailIfQuiesce()
- MQConnectionFactory.getFailIfQuiesce()

Values

YES

Calls to certain methods fail if either the queue manager is in a quiescing state, or the channel being used to connect to a queue manager is quiescing. If an application detects either of these conditions, the application can complete its immediate task and close the connection, allowing the queue manager or channel instance to stop. This is the default value.

NO

No method call fails because either the queue manager, or the channel being used to connect to a queue manager, is in a quiescing state. If you specify this value, an application cannot detect that the queue manager or channel is quiescing. The application might continue to perform operations against the queue manager, and therefore prevent the queue manager from stopping.

HOSTNAME

For a connection to a queue manager, the host name or IP address of the system on which the queue manager is running or, for a real-time connection to a broker, the host name or IP address of the system on which the broker is running.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: HOSTNAME

JMS administration tool short name: HOST

Programmatic access

Setters/getters

- MQConnectionFactory.setHostName()
- MQConnectionFactory.getHostName()

Values

localhost

This is the default value.

Any valid string

LOCALADDRESS

For a connection to a queue manager, this property specifies either the local network interface to be used, or the local port, or range of local ports, to be used.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: LOCALADDRESS

JMS administration tool short name: LA

Programmatic access

Setters/getters

- MQConnectionFactory.setLocalAddress()
- MQConnectionFactory.getLocalAddress()

Values

"" (empty string)

This is the default value.

A string in the format [ip-addr]([low-port[,high-port]])

Here are some examples:

192.0.2.0

The channel binds to address 192.0.2.0 locally.

192.0.2.0(1000)

The channel binds to address 192.0.2.0 locally and uses port 1000.

192.0.2.0(1000,2000)

The channel binds to address 192.0.2.0 locally and uses a port in the range 1000 to 2000.

(1000)

The channel binds to port 1000 locally.

(1000,2000)

The channel binds to a port in the range 1000 to 2000 locally.

You can specify a host name instead of an IP address. For a real-time connection to a broker, this property is relevant only when multicast is used, and the value of the property must not contain a port number, or a range of port numbers. The only valid values of the property in this case are null, an IP address, or a host name.

MAPNAMESTYLE

Allows compatibility style to be used for MapMessage element names.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: MAPNAMESTYLE

JMS administration tool short name: MNST

Programmatic access

Setters/getters

- `MQConnectionFactory.setMapNameStyle()`
- `MQConnectionFactory.getMapNameStyle()`

Values

STANDARD

The standard `com.ibm.jms.JMSMapMessage` element naming format is to be used. This is the default value and allows non-legal Java identifiers to be used as the element name.

COMPATIBLE

The older `com.ibm.jms.JMSMapMessage` element naming format is to be used. Only legal Java identifiers can be used as the element name. This is needed only if map messages are being sent to an application that is using a version of IBM MQ classes for JMS earlier than Version 5.3.

MAXBUFFSIZE

The maximum number of received messages that can be stored in an internal message buffer while waiting to be processed by the application. This property applies only when `TRANSPORT` has the value `DIRECT` or `DIRECTHTTP`.

Applicable Objects

`ConnectionFactory`, `TopicConnectionFactory`

JMS administration tool long name: `MAXBUFFSIZE`

JMS administration tool short name: `MBSZ`

Programmatic access

Setters/getters

- `MQConnectionFactory.setMaxBufferSize()`
- `MQConnectionFactory.getMaxBufferSize()`

Values

1000

This is the default value.

Any positive integer

MDREAD

This property determines whether a JMS application can extract the values of MQMD fields.

Applicable Objects

JMS administration tool long name: `MDREAD`

JMS administration tool short name: `MDR`

Programmatic access

Setters/getters

- `MQDestination.setMQMDReadEnabled()`
- `MQDestination.getMQMDReadEnabled()`

Values

NO

When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD. When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them. This is the default value for administrative tools.

For programs, use False.

Yes

When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including the properties that the sender did not set explicitly. When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including the properties that the sender did not set explicitly.

For programs, use True.

MDWRITE

This property determines whether a JMS application can set the values of MQMD fields.

Applicable Objects

Queue, Topic

JMS administration tool long name: MDWRITE

JMS administration tool short name: MDR

Programmatic access

Setters/getters

- MQDestination.setMQMDWriteEnabled()
- MQDestination.getMQMDWriteEnabled()

Values

NO

All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure. This is the default value for administrative tools.

For programs, use False.

YES

JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure.

For programs, use True.

MDMSGCTX

What level of message context is to be set by the JMS application. The application must be running with appropriate context authority for this property to take effect.

Applicable Objects

JMS administration tool long name: MDMSGCTX

JMS administration tool short name: MDCTX

Programmatic access

Setters/getters

- `MQDestination.setMQMDMessageContext()`
- `MQDestination.getMQMDMessageContext()`

Values

DEFAULT

The MQOPEN API call and the MQPMO structure specify no explicit message context options. This is the default value for administrative tools.

For programs, use `WMQ_MDCTX_DEFAULT`.

SET_IDENTITY_CONTEXT

The MQOPEN API call specifies the message context option `MQOO_SET_IDENTITY_CONTEXT` and the MQPMO structure specifies `MQPMO_SET_IDENTITY_CONTEXT`.

For programs, use `WMQ_MDCTX_SET_IDENTITY_CONTEXT`.

SET_ALL_CONTEXT

The MQOPEN API call specifies the message context option `MQOO_SET_ALL_CONTEXT` and the MQPMO structure specifies `MQPMO_SET_ALL_CONTEXT`.

For programs, use `WMQ_MDCTX_SET_ALL_CONTEXT`.

MSGBATCHSZ

The maximum number of messages to be taken from a queue in one packet when using asynchronous message delivery.

Applicable Objects

`ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`, `XAConnectionFactory`, `XAQueueConnectionFactory`, `XATopicConnectionFactory`

JMS administration tool long name: `MAXBUFFSIZE`

JMS administration tool short name: `MBSZ`

Programmatic access

Setters/getters

- `MQConnectionFactory.setMsgBatchSize()`
- `MQConnectionFactory.getMsgBatchSize()`

Values

10

This is the default value.

Any positive integer

MSGBODY

Determines whether a JMS application accesses the `MQRFH2` of an IBM MQ message as part of the message payload.

Applicable Objects

`Queue`, `Topic`

JMS administration tool long name: WMQ_MESSAGE_BODY

JMS administration tool short name: MBODY

Programmatic access

Setters/getters

- MQConnectionFactory.setMessageBodyStyle()
- MQConnectionFactory.getMessageBodyStyle()

Values

UNSPECIFIED

When sending, IBM MQ classes for JMS does or does not generate and include an MQRFH2 header, depending on the value of WMQ_TARGET_CLIENT. When receiving, acts as value JMS.

JMS

When sending, IBM MQ classes for JMS automatically generates an MQRFH2 header and includes it in the IBM MQ message.

When receiving, IBM MQ classes for JMS set the JMS message properties according to values in the MQRFH2 (if present); it does not present the MQRFH2 as part of the JMS message body.

MQ

When sending, IBM MQ classes for JMS does not generate an MQRFH2.

When receiving, IBM MQ classes for JMS presents the MQRFH2 as part of the JMS message body.

MSGRETENTION

Whether the connection consumer keeps undelivered messages on the input queue.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory,

JMS administration tool long name: MSGRETENTION

JMS administration tool short name: MRET

Programmatic access

Setters/getters

- MQConnectionFactory.setMessageRetention()
- MQConnectionFactory.getMessageRetention()

Values

Yes

Undelivered messages remain on the input queue. This is the default value.

No

Undelivered messages are dealt with according to their disposition options.

MSGSELECTION

Determines whether message selection is done by the IBM MQ classes for JMS or by the broker. If TRANSPORT has the value DIRECT, message selection is always done by the broker and the value of

MSGSELECTION is ignored. Message selection by the broker is not supported when BROKERVER has the value V1.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: MSGSELECTION

JMS administration tool short name: MSEL

Programmatic access

Setters/getters

- MQConnectionFactory.setMessageSelection()
- MQConnectionFactory.getMessageSelection()

Values

CLIENT

Message selection is done by IBM MQ classes for JMS. This is the default value.

BROKER

Message selection is done by the broker.

MULTICAST

To enable multicast on a real-time connection to a broker and, if enabled, to specify the precise way in which multicast is used to deliver messages from the broker to a message consumer. The property has no effect on how a message producer sends messages to a broker.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, Topic

JMS administration tool long name: MULTICAST

JMS administration tool short name: MCAST

Programmatic access

Setters/getters

- MQConnectionFactory.setMulticast()
- MQConnectionFactory.getMulticast()

Values

DISABLED

Messages are not delivered to a message consumer using multicast transport. This is the default value for ConnectionFactory and TopicConnectionFactory objects.

ASCF

Messages are delivered to a message consumer according to the multicast setting for the connection factory associated with the message consumer. The multicast setting for the connection factory is noted at the time that the message consumer is created. This value is valid only for Topic objects, and is the default value for Topic objects.

ENABLED

If the topic is configured for multicast in the broker, messages are delivered to a message consumer using multicast transport. A reliable quality of service is used if the topic is configured for reliable multicast.

RELIABLE

If the topic is configured for reliable multicast in the broker, messages are delivered to the message consumer using multicast transport with a reliable quality of service. If the topic is not configured for reliable multicast, you cannot create a message consumer for the topic.

NOTR

If the topic is configured for multicast in the broker, messages are delivered to the message consumer using multicast transport. A reliable quality of service is not used even if the topic is configured for reliable multicast.

OPTIMISTICPUBLICATION

This property determines whether IBM MQ classes for JMS returns control immediately to a publisher that has published a message, or whether it returns control only after it has completed all the processing associated with the call and can report the outcome to the publisher.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: OPTIMISTICPUBLICATION

JMS administration tool short name: OPTPUB

Programmatic access

Setters/getters

- MQConnectionFactory.setOptimisticPublication()
- MQConnectionFactory.getOptimisticPublication()

Values

NO

When a publisher publishes a message, IBM MQ classes for JMS do not return control to the publisher until it has completed all the processing associated with the call and can report the outcome to the publisher. This is the default value.

YES

When a publisher publishes a message, IBM MQ classes for JMS returns control to the publisher immediately, before it has completed all the processing associated with the call and can report the outcome to the publisher. IBM MQ classes for JMS reports the outcome only when the publisher commits the message.

OUTCOMENOTIFICATION

This property determines whether IBM MQ classes for JMS return control immediately to a subscriber that has just acknowledged or committed a message, or whether it returns control only after it has completed all the processing associated with the call and can report the outcome to the subscriber.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: OUTCOMENOTIFICATION

JMS administration tool short name: NOTIFY

Programmatic access

Setters/getters

- `MQConnectionFactory.setOutcomeNotification()`
- `MQConnectionFactory.getOutcomeNotification()`

Values

YES

When a subscriber acknowledges or commits a message, IBM MQ classes for JMS do not return control to the subscriber until it has completed all the processing associated with the call and can report the outcome to the subscriber. This is the default value.

NO

When a subscriber acknowledges or commits a message, IBM MQ classes for JMS returns control to the subscriber immediately, before it has completed all the processing associated with the call and can report the outcome to the subscriber.

PERSISTENCE

The persistence of messages sent to a destination.

Applicable Objects

Queue, Topic

JMS administration tool long name: PERSISTENCE

JMS administration tool short name: PER

Programmatic access

Setters/getters

- `MQDestination.setPersistence()`
- `MQDestination.getPersistence()`

Values

APP

Persistence is defined by the JMS application. This is the default value.

QDEF

Persistence takes the value of the queue default.

PERS

Messages are persistent.

NON

Messages are nonpersistent.

HIGH

See [JMS persistent messages](#) for further information on the use of this value.

POLLINGINT

If each message listener within a session has no suitable message on its queue, this is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if `TRANSPORT` has the value `BIND` or `CLIENT`.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: POLLINGINT

JMS administration tool short name: PINT

Programmatic access

Setters/getters

- MQConnectionFactory.setPollingInterval()
- MQConnectionFactory.getPollingInterval()

Values

5000

This is the default value.

Any positive integer

PORT

For a connection to a queue manager, the number of the port on which the queue manager is listening or, for a real-time connection to a broker, the number of the port on which the broker is listening for real-time connections.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: PORT

JMS administration tool short name: PORT

Programmatic access

Setters/getters

- MQConnectionFactory.setPort()
- MQConnectionFactory.getPort()

Values

1414

This is the default value if TRANSPORT is set to CLIENT.

1506

This is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.

Any positive integer

PRIORITY

The priority for messages sent to a destination.

Applicable Objects

Queue, Topic

JMS administration tool long name: PRIORITY

JMS administration tool short name: PRI

Programmatic access

Setters/getters

- `MQDestination.setPriority()`
- `MQDestination.getPriority()`

Values

APP

Priority is defined by the JMS application. This is the default value.

QDEF

Priority takes the value of the queue default.

Any integer in the range 0-9

Lowest to highest.

PROCESSDURATION

This property determines whether a subscriber guarantees to process quickly any message it receives before returning control to IBM MQ classes for JMS.

Applicable Objects

`ConnectionFactory`, `TopicConnectionFactory`

JMS administration tool long name: PROCESSDURATION

JMS administration tool short name: PROCDUR

Programmatic access

Setters/getters

- `MQConnectionFactory.setProcessDuration()`
- `MQConnectionFactory.getProcessDuration()`

Values

UNKNOWN

A subscriber can give no guarantee about how quickly it can process any message it receives. This is the default value.

SHORT

A subscriber guarantees to process quickly any message it receives before returning control to IBM MQ classes for JMS.

PROVIDERVERSION

This property differentiates between the three IBM MQ messaging modes of operation: IBM MQ messaging provider normal mode, IBM MQ messaging provider normal mode with restrictions, and IBM MQ messaging provider migration mode.

The IBM MQ messaging provider normal mode uses all the features of an IBM MQ queue manager to implement JMS. This mode is optimized to use the JMS 2.0 API and functionality. The IBM MQ messaging provider normal mode with restrictions uses the JMS 2.0 API, but not the new features such as shared subscriptions, delayed delivery, or asynchronous send.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: PROVIDERVERSION

JMS administration tool short name: PVER

Programmatic access

Setters/getters

- MQConnectionFactory.setProviderVersion()
- MQConnectionFactory.getProviderVersion()

Values

You can set the **PROVIDERVERSION** property to any of the values 8 (normal mode), 7 (normal mode with restrictions), 6 (migration mode), or unspecified (the default value). The value that you specify for the **PROVIDERVERSION** property must be a string. If you are specifying an option of 8, 7 or 6, you can do this in any of the following formats:

- V.R.M.F
- V.R.M
- V.R
- V

where V, R, M and F are integer values greater than or equal to zero. The extra R, M and F values are optional and are available for you to use in case fine grained control is needed. For example, if you wanted to use a **PROVIDERVERSION** level of 7, you could set **PROVIDERVERSION=7**, **7.0**, **7.0.0** or **7.0.0.0**.

8 - Normal mode

The JMS application uses the IBM MQ messaging provider normal mode. Normal mode uses all the features of a IBM MQ queue manager to implement JMS. This mode is optimized to use the JMS 2.0 API and functionality.

If you are connecting to a queue manager with a command level of 800, then all of the JMS 2.0 API and features, such as asynchronous send, delayed delivery, or shared subscription, can be used.

If the queue manager specified in the connection factory settings is not a Version 8.0.0 queue manager, the `createConnection` method fails with an exception JMSFMQ0003.

The IBM MQ messaging provider normal mode uses the sharing conversations feature and the number of conversations that can be shared is controlled by the **SHARECNV()** property on the server connection channel. If this property is set to 0, you cannot use IBM MQ messaging provider normal mode and the `createConnection` method fails with an exception JM5CC5007.

7 - Normal mode with restrictions

The JMS application uses the IBM MQ messaging provider normal mode with restrictions. This mode uses the JMS 2.0 API, but not the new features such as shared subscriptions, delayed delivery, or asynchronous send.

If you set **PROVIDERVERSION** to 7 only the IBM MQ messaging provider normal with restrictions mode of operation is available. If the queue manager specified in the connection factory settings is not a Version 7.0.1, or later, queue manager, the `createConnection` method fails with exception JM5FCC5008.

If you are connecting using normal mode with restrictions, to a queue manager with a command level between 700 and 800 then you can use the JMS 2.0 API, but not the asynchronous send, delayed delivery, or shared subscription features.

The IBM MQ messaging provider normal mode with restrictions uses the sharing conversations feature and the number of conversations that can be shared is controlled by the **SHARECNV()** property on the server connection channel. If this property is set to 0, you cannot use IBM MQ messaging provider normal mode with restrictions and the `createConnection` method fails with an exception JMSCC5007.

6 - Migration mode

The JMS application uses the IBM MQ messaging provider migration mode.

The IBM MQ classes for JMS use the features and algorithms supplied with IBM WebSphere MQ 6.0. If you want to connect to WebSphere Message Broker Version 6.0 or 6.1 using IBM WebSphere MQ Enterprise Transport Version 6.0, you must use this mode. You can connect to a IBM MQ 8.0 queue manager using this mode, but none of the new features of a IBM MQ classes for JMS queue manager are used, for example, read ahead or streaming.

If you have a IBM MQ 8.0 client connecting to a IBM MQ 8.0 queue manager, then the message selection is done by the queue manager rather than on the client system.

If IBM MQ messaging provider migration mode is specified and you attempt to use any of the JMS 2.0 API, the API method call fails with the exception JMSCC5007.

unspecified (default)

The **PROVIDERVERSION** property is set to *unspecified* by default.

A connection factory that was created with a previous version of IBM MQ classes for JMS in JNDI takes this value when the connection factory is used with the new version of IBM MQ classes for JMS. The following algorithm is used to determine which mode of operation is used. This algorithm is used when the `createConnection` method is called and uses other aspects of the connection factory to determine if IBM MQ messaging provider normal mode, normal mode with restrictions, or IBM MQ messaging provider migration mode is required.

1. First, an attempt to use IBM MQ messaging provider normal mode is made.
2. If the queue manager connected is not IBM MQ 8.0, or later, an attempt to use IBM MQ messaging provider normal mode with restrictions is made.
3. If the queue manager connected is not IBM WebSphere MQ 7.0.1, or later, the connection is closed and IBM MQ messaging provider migration mode is used instead.
4. If the **SHARECNV** property on the server connection channel is set to 0, the connection is closed and IBM MQ messaging provider migration mode is used instead.
5. If **BROKERVER** is set to V1 or the default *unspecified* value, IBM MQ messaging provider normal mode continues to be used, and therefore any publish/subscribe operations use the new IBM WebSphere MQ 7.0.1, or later, features.

See [ALTER QMGR](#) for information about the **PSMODE** parameter of the **ALTER QMGR** command for further information on compatibility.

6. If **BROKERVER** is set to V2 the action taken depends on the value of **BROKERQMGR** :

- If the **BROKERQMGR** is blank:

If the queue specified by the **BROKERCONQ** property can be opened for output (that is, **MQOPEN** for output succeeds) and **PSMODE** on the queue manager is set to **COMPAT** or **DISABLED**, then IBM MQ messaging provider migration mode is used.

- If the queue specified by the **BROKERCONQ** property cannot be opened for output, or the **PSMODE** attribute is set to **ENABLED**:

IBM MQ messaging provider normal mode is used.

- If **BROKERQMGR** is non-blank :

IBM MQ messaging provider migration mode is used.

If you cannot change the connection factory that you are using, you can use the `com.ibm.msg.client.wmq.overrideProviderVersion` property to override any setting on the

connection factory. This override applies to all connection factories in the JVM but the actual connection factory objects are not modified.

Related concepts

[Rules for selecting the IBM MQ messaging provider mode](#)

Related tasks

[Configuring the JMS **PROVIDERVERSION** property](#)

PROXYHOSTNAME

The host name or IP address of the system on which the proxy server is running when using a real-time connection to a broker through a proxy server.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: PROXYHOSTNAME

JMS administration tool short name: PHOST

Programmatic access

Setters/getters

- MQConnectionFactory.setProxyHostName()
- MQConnectionFactory.getProxyHostName()

Values

null

The host name of the proxy server. This is the default value.

PROXYPORT

The number of the port on which the proxy server is listening when using a real-time connection to a broker through a proxy server.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: PROXYPORT

JMS administration tool short name: PPORT

Programmatic access

Setters/getters

MQConnectionFactory.setProxyPort()

MQConnectionFactory.getProxyPort()

Values

443

The port number of the proxy server. This is the default value.

PUBACKINT

The number of messages published by a publisher before IBM MQ classes for JMS requests an acknowledgment from the broker.

When you lower the value of this property, IBM MQ classes for JMS requests acknowledgments more often, therefore the performance of the publisher decreases. When you raise the value, IBM MQ classes for JMS take a longer time to throw an exception if the broker fails. This property is relevant only if TRANSPORT has the value BIND or CLIENT.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: PROXYPORT

JMS administration tool short name: PPORT

Programmatic access

Setters/getters

MQConnectionFactory.setPubAckInterval()

MQConnectionFactory.getPubAckInterval()

Values

25

Any positive integer may be the default value.

PUTASYNCALLOWED

This property determines whether message producers are allowed to use asynchronous puts to send messages to this destination.

Applicable Objects

Queue, Topic

JMS administration tool long name: PUTASYNCALLOWED

JMS administration tool short name: PAALD

Programmatic access

Setters/getters

MQDestination.setPutAsyncAllowed()

MQDestination.getPutAsyncAllowed()

Values

AS_DEST

Determine whether asynchronous puts are allowed by referring to the queue or topic definition. This is the default value.

AS_Q_DEF

Determine whether asynchronous puts are allowed by referring to the queue definition.

AS_TOPIC_DEF

Determine whether asynchronous puts are allowed by referring to the topic definition.

NO

Asynchronous puts are not allowed.

YES

Asynchronous puts are allowed.

QMANAGER

The name of the queue manager to connect to.

However, if your application uses a client channel definition table to connect to a queue manager, see [Using a client channel definition table with IBM MQ classes for JMS](#).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, Queue, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: QMANAGER

JMS administration tool short name: QMGR

Programmatic access

Setters/getters

- MQConnectionFactory.setQueueManager()
- MQConnectionFactory.getQueueManager()

Values

" " (empty string)

Any string can be the default value.

QUEUE

The name of the JMS queue destination. This matches the name of the queue used by the queue manager.

Applicable Objects

Queue

JMS administration tool long name: QUEUE

JMS administration tool short name: QU

Values

Any string

Any valid IBM MQ queue name.

Related concepts

[Rules for naming IBM MQ objects](#)>

READAHEADALLOWED

This property determines whether message consumers and queue browsers are allowed to use read ahead to get nonpersistent messages from this destination into an internal buffer before receiving them.

Applicable Objects

Queue, Topic

JMS administration tool long name: READAHEADALLOWED

JMS administration tool short name: RAALD

Programmatic access

Setters/getters

- `MQDestination.setReadAheadAllowed()`
- `MQDestination.getReadAheadAllowed()`

Values

AS_DEST

Determine whether read ahead is allowed by referring to the queue or topic definition. This is the default value in administrative tools.

Use `WMQConstants.WMQ_READ_AHEAD_ALLOWED_AS_DEST` in programs.

AS_Q_DEF

Determine whether read ahead is allowed by referring to the queue definition.

Use `WMQConstants.WMQ_READ_AHEAD_ALLOWED_AS_Q_DEF` in programs.

AS_TOPIC_DEF

Determine whether read ahead is allowed by referring to the topic definition.

Use `WMQConstants.WMQ_READ_AHEAD_ALLOWED_AS_TOPIC_DEF` in programs.

NO

Read ahead is not allowed.

Use `WMQConstants.WMQ_READ_AHEAD_ALLOWED_DISABLED` in programs.

YES

Read ahead is allowed.

Use `WMQConstants.WMQ_READ_AHEAD_ALLOWED_ENABLED` in programs.

READAHEADCLOSEPOLICY

For messages being delivered to an asynchronous message listener, what happens to messages in the internal read ahead buffer when the message consumer is closed.

Applicable Objects

Queue, Topic

JMS administration tool long name: READAHEADCLOSEPOLICY

JMS administration tool short name: RACP

Programmatic access

Setters/getters

- `MQDestination.setReadAheadClosePolicy()`

- `MQDestination.getReadAheadClosePolicy()`

Values

DELIVER_ALL

All messages in the internal read ahead buffer are delivered to the message listener of the application before returning. This is the default value in administrative tools.

Use `WMQConstants.WMQ_READ_AHEAD_DELIVERALL` in programs.

DELIVER_CURRENT

Only the current message listener invocation completes before returning, potentially leaving messages in the internal read ahead buffer, which are then discarded.

Use `WMQConstants.WMQ_READ_AHEAD_DELIVERCURRENT` in programs.

RECEIVECCSID

Destination property that sets the target CCSID for queue manager message conversion. The value is ignored unless `RECEIVECONVERSION` is set to `WMQ_RECEIVE_CONVERSION_QMGR`

Applicable Objects

Queue, Topic

JMS administration tool long name: `RECEIVECCSID`

JMS administration tool short name: `RCCS`

Programmatic access

Setters/Getters

- `MQDestination.setReceiveCCSID`
- `MQDestination.getReceiveCCSID`

Values

WMQConstants.WMQ_RECEIVE_CCSID_JVM_DEFAULT

0 - Use `JVM Charset.defaultCharset`

1208

UTF-8

CCSID

Supported coded character set identifier.

RECEIVECONVERSION

Destination property that determines if data conversion is going to be performed by the queue manager.

Applicable Objects

Queue, Topic

JMS administration tool long name: `RECEIVECONVERSION`

JMS administration tool short name: `RCNV`

Programmatic access

Setters/Getters

- `MQDestination.setReceiveConversion`

- `MQDestination.getReceiveConversion`

Values

WMQConstants.WMQ_RECEIVE_CONVERSION_CLIENT_MSG

1 - Only perform data conversion on the JMS client. The default value from up to V7.0, and from, and including, 7.0.1.5.

WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR

2 - Perform data conversion on the queue manager before sending a message to the client. The default (and only) value from V7.0 to V7.0.1.4 inclusive, except if APAR IC72897 is applied.

RECEIVEISOLATION

This property determines whether a subscriber might receive messages that have not been committed on the subscriber queue.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: RECEIVEISOLATION

JMS administration tool short name: RCVISOL

Values

COMMITTED

A subscriber receives only those messages on the subscriber queue that have been committed. This is the default value in administrative tools.

Use `WMQConstants.WMQ_RCVISOL_COMMITTED` in programs.

UNCOMMITTED

A subscriber can receive messages that have not been committed on the subscriber queue.

Use `WMQConstants.WMQ_RCVISOL_UNCOMMITTED` in programs.

RECEXIT

Identifies a channel receive exit, or a sequence of receive exits to be run in succession.

Additional configuration might be required in order for the IBM MQ classes for JMS to locate receive exits. For more information, see [Configuring the IBM MQ classes for JMS to use channel exits](#).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: RECEXIT

JMS administration tool short name: RCX

Programmatic access

Setters/getters

- `MQConnectionFactory.setReceiveExit()`
- `MQConnectionFactory.getReceiveExit()`

Values

null

A string comprising one or more items separated by commas, where each item is either:

- The name of a class that implements the WMQReceiveExit interface (for a channel receive exit written in Java).
- A string in the format *libraryName(entryPointName)* (for a channel receive exit not written in Java).

This is the default value.

RECEXITINIT

The user data that is passed to channel receive exits when they are called.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: RECEXITINIT

JMS administration tool short name: RCXI

Programmatic access

Setters/getters

- MQConnectionFactory.setReceiveExitInit()
- MQConnectionFactory.getReceiveExitInit()

Values

null

A string comprising one or more items of user data separated by commas. This is the default value.

REPLYTOSTYLE

Determines how the JMSReplyTo field in a received message is constructed.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: REPLYTOSTYLE

JMS administration tool short name: RTOST

Programmatic access

Setters/getters

- MQConnectionFactory.setReplyToStyle()
- MQConnectionFactory.getReplyToStyle()

Values

DEFAULT

Equivalent to MQMD.

RFH2

Use the value supplied in the RFH2 header. If a JMSReplyTo value has been set in the sending application, use that value.

MQMD

Use the MQMD supplied value. This behavior is equivalent to the default behavior of IBM WebSphere MQ 6.0.2.4 and 6.0.2.5.

If the JMSReplyTo value set by the sending application does not contain a queue manager name, the receiving queue manager inserts its own name in the MQMD. If you set this parameter to MQMD, the reply-to queue you use is on the receiving queue manager. If you set this parameter to RFH2, the reply-to queue you use is on the queue manager specified in the RFH2 of the sent message as originally set by the sending application.

If the JMSReplyTo value set by the sending application contains a queue manager name, the value of this parameter is unimportant because both the MQMD and RFH2 contain the same value.

RESCANINT

When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, IBM MQ classes for JMS search the IBM MQ queue for suitable messages in the sequence determined by the `MsgDeliverySequence` attribute of the queue.

After IBM MQ classes for JMS find a suitable message and deliver it to the consumer, IBM MQ classes for JMS resume the search for the next suitable message from its current position in the queue. IBM MQ classes for JMS continue to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, IBM MQ classes for JMS return to the beginning of the queue to continue the search, and a new time interval commences.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory

JMS administration tool long name: RESCANINT

JMS administration tool short name: RINT

Programmatic access

Setters/getters

- `MQConnectionFactory.setRescanInterval()`
- `MQConnectionFactory.getRescanInterval()`

Values**5000**

Any positive integer can be the default value.

SECEXIT

Identifies a channel security exit.

Additional configuration might be required in order for the IBM MQ classes for JMS to locate security exits. For more information, see [Configuring the IBM MQ classes for JMS to use channel exits](#).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SECEXIT

JMS administration tool short name: SXC

Programmatic access

Setters/getters

- MQConnectionFactory.setSecurityExit()
- MQConnectionFactory.getSecurityExit()

Values

null

The name of a class that implements the WMQSecurityExit interface (for a channel security exit written in Java).

A string in the format *libraryName(entryPointName)* (for a channel security exit not written in Java).

SECEXITINIT

The user data that is passed to a channel security exit when it is called.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SECEXITINIT

JMS administration tool short name: SCXI

Programmatic access

Setters/getters

- MQConnectionFactory.setSecurityExitInit()
- MQConnectionFactory.getSecurityExitInit()

Values

null

Any string can be the default value.

SENDCHECKCOUNT

The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted JMS session.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SENDCHECKCOUNT

JMS administration tool short name: SCC

Programmatic access

Setters/getters

- MQConnectionFactory.setSendCheckCount()
- MQConnectionFactory.getSendCheckCount()

Values

null

Any string can be the default value.

SENDEXIT

Identifies a channel send exit, or a sequence of send exits to be run in succession.

Additional configuration might be required in order for the IBM MQ classes for JMS to locate send exits. For more information, see [Configuring the IBM MQ classes for JMS to use channel exits](#).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SENDEXIT

JMS administration tool short name: SDX

Programmatic access

Setters/getters

- MQConnectionFactory.setSendExit()
- MQConnectionFactory.getSendExit()

Values

null

Any string comprising one or more items separated by commas, where each item is either:

- The name of a class that implements the WMQSendExit interface (for a channel send exit written in Java).
- A string in the format *libraryName(entryPointName)* (for a channel send exit not written in Java).
-

This is the default value.

SENDEXITINIT

The user data that is passed to channel send exits when they are called.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SENDEXITINIT

JMS administration tool short name: SDXI

Programmatic access

Setters/getters

- MQConnectionFactory.setSendExitInit()

- MQConnectionFactory.getSendExitInit()

Values

null

Any string comprising one or more items of user data separated by commas can be the default value.

SHARECONVALLOWED

This property determines whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SHARECONVALLOWED

JMS administration tool short name: SCALD

Programmatic access

Setters/getters

- MQConnectionFactory.setShareConvAllowed()
- MQConnectionFactory.getShareConvAllowed()

Values

YES

This is the default value for administrative tools.

For programs, use WMQConstants.WMQ_SHARE_CONV_ALLOWED_YES.

NO

This value is for administrative tools.

For programs, use WMQConstants.WMQ_SHARE_CONV_ALLOWED_NO.

SPARSESUBS

Controls the message retrieval policy of a TopicSubscriber object.

Applicable Objects

ConnectionFactory, TopicConnectionFactory

JMS administration tool long name: SPARSESUBS

JMS administration tool short name: SSUBS

Programmatic access

Setters/getters

- MQConnectionFactory.setSparseSubscriptions()
- MQConnectionFactory.getSparseSubscriptions()

Values

NO

Subscriptions receive frequent matching messages. This is the default value for administrative tools.

For programs, use false.

YES

Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.

For programs, use true.

SSLCIPHERSUITE

The CipherSuite to use for an SSL connection.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SSLCIPHERSUITE

JMS administration tool short name: SCPHS

Programmatic access

Setters/getters

- MQConnectionFactory.setSSLCipherSuite()
- MQConnectionFactory.getSSLCipherSuite()

Values

null

This is the default value. For more information, see [SSL properties of JMS objects](#).

SSLCRL

CRL servers to check for SSL certificate revocation.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SSLCRL

JMS administration tool short name: SCRL

Programmatic access

Setters/getters

- MQConnectionFactory.setSSLCertStores()
- MQConnectionFactory.getSSLCertStores()

Values

null

Space-separated list of LDAP URLs. This is the default value. For more information, see [SSL properties of JMS objects](#).

SSLFIPSREQUIRED

This property determines whether an SSL connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS).

Note: On UNIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the [IBM Crypto for C \(ICC\) certificate](#) and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the [NIST CMVP modules in process list](#).

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SSLFIPSREQUIRED

JMS administration tool short name: SFIPS

Programmatic access

Setters/getters

- MQConnectionFactory.setSSLFipsRequired()
- MQConnectionFactory.getSSLFipsRequired()

Values

NO

An SSL connection can use any CipherSuite that is not supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS).

This is the default value. In programs, use false.

YES

An SSL connection must use a CipherSuite that is supported by IBMJSSEFIPS.

In programs, use true.

SSLPEERNAME

For SSL, a *distinguished name* skeleton that must match that provided by the queue manager.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SSLPEERNAME

JMS administration tool short name: SPEER

Programmatic access

Setters/getters

- MQConnectionFactory.setSSLPeerName()
- MQConnectionFactory.getSSLPeerName()

Values

null

This is the default value. For more information, see [SSL properties of JMS objects](#).

SSLRESETCOUNT

For SSL, the total number of bytes sent and received by a connection before the secret key that is used for encryption is renegotiated.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SSLRESETCOUNT

JMS administration tool short name: SRC

Programmatic access

Setters/getters

- MQConnectionFactory.setSSLResetCount()
- MQConnectionFactory.getSSLResetCount()

Values

0

Zero, or any positive integer less than or equal to 999, 999, 999. This is the default value. For more information, see [SSL properties of JMS objects](#).

STATREFRESHINT

The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager.

This property is relevant only if SUBSTORE has the value QUEUE.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: STATREFRESHINT

JMS administration tool short name: SRI

Programmatic access

Setters/getters

- MQConnectionFactory.setStatusRefreshInterval()
- MQConnectionFactory.getStatusRefreshInterval()

Values

60000

Any positive integer can be the default value. For more information, see [SSL properties of JMS objects](#).

SUBSTORE

Where IBM MQ classes for JMS stores persistent data relating to active subscriptions.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SUBSTORE

JMS administration tool short name: SS

Programmatic access

Setters/getters

- MQConnectionFactory.setSubscriptionStore()
- MQConnectionFactory.getSubscriptionStore()

Values

BROKER

Use the broker-based subscription store to hold details of subscriptions. This is the default value for administrative tools.

For programs, use WMQConstants.WMQ_SUBSTORE_BROKER.

MIGRATE

Transfer subscription information from the queue-based subscription store to the broker-based subscription store.

For programs, use WMQConstants.WMQ_SUBSTORE_MIGRATE.

QUEUE

Use the queue-based subscription store to hold details of subscriptions.

For programs, use WMQConstants.WMQ_SUBSTORE_QUEUE.

SYNCPOINTALLGETS

This property determines whether all gets are to be performed under syncpoint.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: SYNCPOINTALLGETS

JMS administration tool short name: SPAG

Programmatic access

Setters/getters

- MQConnectionFactory.setSyncpointAllGets()
- MQConnectionFactory.getSyncpointAllGets()

Values

No

This is the default value.

Yes

TARGCLIENT

This property determines whether the IBM MQ RFH2 format is used to exchange information with target applications.

Applicable Objects

Queue, Topic

JMS administration tool long name: TARGCLIENT

JMS administration tool short name: TC

Programmatic access

Setters/getters

- `MQDestination.setTargetClient()`
- `MQDestination.getTargetClient()`

Values

JMS

The target of the message is a JMS application. This is the default value for administrative tools.

For programs, use `WMQConstants.WMQ_CLIENT_JMS_COMPLIANT`.

MQ

The target of the message is a non-JMS IBM MQ application.

For programs, use `WMQConstants.WMQ_CLIENT_NONJMS_MQ`.

TARGCLIENTMATCHING

This property determines whether a reply message, sent to the queue identified by the `JMSReplyTo` header field of an incoming message, has an `MQRFH2` header only if the incoming message has an `MQRFH2` header.

Applicable Objects

`ConnectionFactory`, `QueueConnectionFactory`, `XAConnectionFactory`, `XAQueueConnectionFactory`

JMS administration tool long name: TARGCLIENTMATCHING

JMS administration tool short name: TCM

Programmatic access

Setters/getters

- `MQConnectionFactory.setTargetClientMatching()`
- `MQConnectionFactory.getTargetClientMatching()`

Values

YES

If an incoming message does not have an `MQRFH2` header, the `TARGCLIENT` property of the `Queue` object derived from the `JMSReplyTo` header field of the message is sent to MQ. If the message does have an `MQRFH2` header, the `TARGCLIENT` property is set to `JMS` instead. This is the default value for administrative tools.

For programs, use `true`.

NO

The TARGCLIENT property of the Queue object derived from the JMSReplyTo header field of an incoming message is always set to JMS.

For programs, use false.

TEMPMODEL

The name of the model queue from which JMS temporary queues are created.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory

JMS administration tool long name: TEMPMODEL

JMS administration tool short name: TM

Programmatic access

Setters/getters

- MQConnectionFactory.setTemporaryModel()
- MQConnectionFactory.getTemporaryModel()

Values

SYSTEM.DEFAULT.MODEL.QUEUE

Any string can be the default value.

TEMPQPREFIX

The prefix that is used to form the name of an IBM MQ dynamic queue.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory

JMS administration tool long name: TEMPQPREFIX

JMS administration tool short name: TQP

Programmatic access

Setters/getters

- MQConnectionFactory.setTempQPrefix()
- MQConnectionFactory.getTempQPrefix()

Values

" " (empty string)

The prefix used is CSQ.* on z/OS and AMQ.* on all other platforms. These are the default values.

queue prefix

The queue prefix is any string that conforms to the rules for forming contents of the *DynamicQName* field in an IBM MQ object descriptor (structure MQOD), but the last non-blank character must be an asterisk.

TEMPTOPICPREFIX

When creating temporary topics, JMS generates a topic string of the form "TEMP /*TEMPTOPICPREFIX*/*unique_id*", or if this property is left with the default value, just "TEMP /*unique_id*". Specifying a non-empty TEMPTOPICPREFIX allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.

Applicable Objects

ConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: TEMPTOPICPREFIX

JMS administration tool short name: TTP

Programmatic access

Setters/getters

- MQConnectionFactory.setTempTopicPrefix()
- MQConnectionFactory.getTempTopicPrefix()

Values

Any non-null string consisting only of valid characters for an IBM MQ topic string. The default value is "" (empty string).

TOPIC

The name of the JMS topic destination, this value is used by the queue manager as the topic string of a publication or subscription.

Applicable Objects

Topic

JMS administration tool long name: TOPIC

JMS administration tool short name: TOP

Values

Any string

A string that forms a valid IBM MQ topic string. When using IBM MQ as a messaging provider with WebSphere Application Server, specify a value that matches the name by which the topic is known for administrative purposes within WebSphere Application Server.

Related concepts

[Topic strings](#)

TRANSPORT

The nature of a connection to a queue manager or broker.

Applicable Objects

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory, XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory

JMS administration tool long name: TRANSPORT

JMS administration tool short name: TRAN

Programmatic access

Setters/getters

- `MQConnectionFactory.setTransportType()`
- `MQConnectionFactory.getTransportType()`

Values

BIND

For a connection to a queue manager in bindings mode. This is the default value for administrative tools.

For programs, use `WMQConstants.WMQ_CM_BINDINGS`.

CLIENT

For a connection to a queue manager in client mode.

For programs, use `WMQConstants.WMQ_CM_CLIENT`.

DIRECT

For a real-time connection to a broker not using HTTP tunnelling.

For programs, use `WMQConstants.WMQ_CM_DIRECT_TCPIP`.

DIRECTHTTP

For a real-time connection to a broker using HTTP tunnelling. Only HTTP 1.0 is supported.

For programs, use `WMQConstants.WMQ_CM_DIRECT_HTTP`.

Related concepts

[“Dependencies between properties of IBM MQ classes for JMS objects” on page 1931](#)

The validity of some properties is dependent on the particular values of other properties.

WILDCARDFORMAT

This property determines which version of wildcard syntax is to be used.

Applicable Objects

`ConnectionFactory`, `TopicConnectionFactory`, `XAConnectionFactory`, `XATopicConnectionFactory`

JMS administration tool long name: `WILDCARDFORMAT`

JMS administration tool short name: `WCFMT`

Programmatic access

Setters/getters

- `MQConnectionFactory.setWildCardFormat()`
- `MQConnectionFactory.getWildCardFormat()`

Values

TOPIC_ONLY

Recognizes topic level wildcards only, as used in broker version 2. This is the default value for administrative tools.

For programs, use `WMQConstants.WMQ_WILDCARD_TOPIC_ONLY`.

CHAR_ONLY

Recognizes character wildcards only, as used in broker version 1.

For programs, use `WMQConstants.WMQ_WILDCARD_CHAR_ONLY`.

The ENCODING property

The ENCODING property comprises three sub-properties, in twelve possible combinations.

The valid values that the ENCODING property can take are constructed from the three sub-properties:

integer encoding

Either normal or reversed

decimal encoding

Either normal or reversed

floating-point encoding

IEEE normal, IEEE reversed, or z/OS

The ENCODING property is expressed as a three-character string with the following syntax:

```
{N|R}{N|R}{N|R|3}
```

In this string:

- N denotes normal
- R denotes reversed
- 3 denotes z/OS
- The first character represents *integer encoding*
- The second character represents *decimal encoding*
- The third character represents *floating-point encoding*

This provides a set of twelve possible values for the ENCODING property.

There is an additional value, the string NATIVE, which sets appropriate encoding values for the Java platform.

The following examples show valid combinations for ENCODING:

```
ENCODING(NNR)  
ENCODING(NATIVE)  
ENCODING(RR3)
```

SSL properties of JMS objects

Enable Secure Sockets Layer (SSL) encryption using the SSLCIPHERSUITE property. You can then change the characteristics of the SSL encryption using several other properties.

When you specify TRANSPORT(CLIENT), you can enable Secure Sockets Layer (SSL) encrypted communication using the SSLCIPHERSUITE property. Set this property to a valid CipherSuite provided by your JSSE provider; it must match the CipherSpec named on the SVRCONN channel named by the CHANNEL property.

However, CipherSpecs (as specified on the SVRCONN channel) and CipherSuites (as specified on ConnectionFactory objects) use different naming schemes to represent the same SSL encryption algorithms. If a recognized CipherSpec name is specified on the SSLCIPHERSUITE property, JMSAdmin issues a warning and maps the CipherSpec to its equivalent CipherSuite. See [SSL/TLS CipherSpecs and CipherSuites in IBM MQ classes for JMS](#) for a list of CipherSpecs recognized by IBM MQ and JMSAdmin.

If you require a connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), set the SSLFIPSREQUIRED property of the connection factory to YES. The default value of this property is NO, which means that a connection can use any supported CipherSuite. The property is ignored if SSLCIPHERSUITE is not set.

The SSLPEERNAME matches the format of the SSLPEER parameter, which can be set on channel definitions. It is a list of attribute name and value pairs separated by commas or semicolons. For example:

```
SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSphere)
```

The set of names and values makes up a *distinguished name*. For more details about distinguished names and their use with IBM MQ, see [Security](#).

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

If SSLPEERNAME is not set, no such checking is performed. SSLPEERNAME is ignored if SSLCIPHERSUITE is not set.

The SSLCRL property specifies zero or more CRL (Certificate Revocation List) servers. Use of this property requires a JVM at Java 2 v1.4. This is a space-delimited list of entries of the form:

```
ldap:// hostname:[ port ]
```

optionally followed by a single /. If *port* is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. See [Security](#) for more about CRL security.

If SSLCRL is not set, no such checking is performed. SSLCRL is ignored if SSLCIPHERSUITE is not set.

The SSLRESETCOUNT property represents the total number of bytes sent and received by a connection before the secret key that is used for encryption is renegotiated. The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by IBM MQ classes for JMS.

For example, to configure a ConnectionFactory object that can be used to create a connection over an SSL enabled MQI channel with a secret key that is renegotiated after 4 MB of data have flowed, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLRESETCOUNT(4194304)
```

If the value of SSLRESETCOUNT is zero, which is the default value, the secret key is never renegotiated. The SSLRESETCOUNT property is ignored if SSLCIPHERSUITE is not set.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of WebSphere MQ.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Important: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks

IBM, the IBM logo, ibm.com[®], are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" www.ibm.com/legal/copytrade.shtml. Other product and service names might be trademarks of IBM or other companies.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

This product includes software developed by the Eclipse Project (<http://www.eclipse.org/>).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.



Part Number:

(1P) P/N: