WebSphere MQ for i5/OS

Application Programming Reference (ILE RPG)

Version 7.0

SC34-6943-00

WebSphere MQ for i5/OS

Application Programming Reference (ILE RPG)

Version 7.0

SC34-6943-00

Note
Before using this information and the product it supports, be sure to read the general information under notices at the back of this book.

First edition (April 2008)

This edition of the book applies to the following:

- IBM WebSphere MQ for i5/OS, Version 7.0

and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Chapter 1. Data type descriptions

Elementary data types

This chapter describes the elementary data types used by the MQI.

The elementary data types are:

- MQBOOL – Boolean
- MQBYTE – Byte
- MQBYTEn – String of n bytes
- MQCHAR – Single-byte character
- MQCHARn – String of n single-byte characters
- MQFLOAT32 – 32-bit floating-point number
- MQFLOAT64 – 64-bit floating point number
- MQHCONN – Connection handle
- MQHOBJ – Object handle
- MQINT8 – 8-bit signed integer
- MQUINT8 – 8-bit unsigned integer
- MQINT16 – 16-bit signed integer
- MQUINT16 – 16-bit unsigned integer
- MQINT64 – 64-bit signed integer
- MQUINT64 – 64-bit unsigned integer
- MQLONG – Long integer
- PMQINT64 – Pointer to data of type MQINT64
- PMQUINT64 – Pointer to data of type MQUINT64

Conventions used in the descriptions of data types

For each elementary data type, this chapter gives a description of its usage, in a form that is independent of the programming language. This is followed by a 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 i5/OS® objects. If an MQ data type definition changes, it is a relatively simple matter to recreate these objects.

Elementary data types

All of the other data types described in this chapter equate either directly to these elementary data types, or to aggregates of these elementary data types (arrays or structures).

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 whose nature is not known to the queue manager. For example, the area may 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 a 8-byte boundary.

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.

Overview for MQHMSG

Purpose: 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

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

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.

MQUINT8

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

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.

MQINT32 – 32 bit integer

The MQINT32 data type is a 32 bit signed integer. It is equivalent to MQLONG.

MQUINT32 – 32 bit unsigned integer

The MQUINT32 data type is a 32 bit unsigned integer. It is equivalent to MQULONG.

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 a 8 byte boundary.

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.

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.

PMQINT32 – Pointer to data of type MQINT32

The PMQINT32 data type is a pointer to data of type MQINT32. It is equivalent to PMQLONG.

PMQUINT32 – Pointer to data of type MQUINT32

The PMQUINT32 data type is a pointer to data of type MQUINT32. It is equivalent to PMQULONG.

PMQINT64 – Pointer to data of type MQINT64

The PMQINT64 data type is a pointer to data of type MQINT64.

PMQUINT64 – Pointer to data of type MQUINT64

The PMQUINT64 data type is a pointer to data of type MQUINT64.

Elementary data types

Table 1. Elementary data types

Data type	Representation
MQBOOL	A 10-digit signed integer.
MQBYTE	A 1-byte alphanumeric field.
MQBYTE16	A 16-byte alphanumeric field.
MQBYTE24	A 24-byte alphanumeric field.
MQBYTE32	A 32-byte alphanumeric field.
MQBYTE64	A 64-byte alphanumeric field.
MQCHAR	A 1-byte alphanumeric field.
MQCHAR4	A 4-byte alphanumeric field.
MQCHAR8	An 8-byte alphanumeric field.
MQCHAR12	A 12-byte alphanumeric field.
MQCHAR16	A 16-byte alphanumeric field.
MQCHAR20	A 20-byte alphanumeric field.
MQCHAR28	A 28-byte alphanumeric field.
MQCHAR32	A 32-byte alphanumeric field.
MQCHAR48	A 48-byte alphanumeric field.
MQCHAR64	A 64-byte alphanumeric field.
MQCHAR128	A 128-byte alphanumeric field.
MQCHAR256	A 256-byte alphanumeric field.
MQFLOAT32	A 4-byte floating-point number.
MQFLOAT64	An 8-byte floating-point number.
MQHCONN	A 10-digit signed integer.
MQHOBJ	A 10-digit signed integer.
MQINT8	A 3-digit signed integer.
MQUINT8	A 3-digit unsigned integer.
MQINT16	A 5-digit signed integer.
MQUINT16	A 5-digit unsigned integer.
MQINT64	A 64-bit signed integer.
MQUINT64	A 64-bit unsigned integer.
MQLONG	A 10-digit signed integer.
PMQINT64	Pointer to data of type MQINT64.
PMQLONG	A 10-digit signed integer.
PMQUINT64	Pointer to data of type MQUINT64.

Language considerations

This section contains information to help you use the MQI from the RPG programming language.

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 8.
- 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 8.

The COPY files reside in QRPGLESRC 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 2.

Table 2. RPG COPY files

Filename (static linkage, initialized, CMQ*G)	Filename (static linkage, not initialized, CMQ*H)	Filename (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
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 mame for CFIN64
CMQCIHG	CMQCIHH	–	CICS® information header structure
CMQCNOG	CMQCNOH	–	Connect options structure

Table 2. RPG COPY files (continued)

Filename (static linkage, initialized, CMQ*G)	Filename (static linkage, not initialized, CMQ*H)	Filename (dynamic linkage, initialized, CMQ*R)	Contents
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
CMQGMOG	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
CMQPMOG	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
CMQRRG	CMQRRH	CMQRRR	Response record structure
CMQTMCG	CMQTMCH	CMQTMCR	Trigger message structure (character format)
CMQTMC2G	CMQTMC2H	CMQTMC2R	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

In this book, the calls are described using their individual names. For calls using dynamic linkage to program QMQM/QMQM, see the *MQSeries for AS/400 V4R2.1 Administration Guide*.

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 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 are recommended to use these named constants and not the values they represent, as this improves the readibility 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 prior to 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 below 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 below for special considerations when writing threaded 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 OS/400® and i5/OS is part of the WebSphere® Development Toolset and WebSphere Development Studio for i5/OS 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 may 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 *i5/OS WebSphere MQ Development Studio: ILE RPG Reference*, and the *i5/OS WebSphere 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.

Coding the bound calls

MQI ILE procedures are listed in Table 3.

Table 3. ILE RPG bound calls supported by each service program

Name of call	LIBMQM and LIBMQM_R	AMQZSTUB	AMQVSTUB
MQBACK	Y		

Table 3. ILE RPG bound calls supported by each service program (continued)

Name of call	LIBMQM and LIBMQM_R	AMQZSTUB	AMQVSTUB
MQBEGIN	Y		
MQCMIT	Y		
MQCLOSE	Y	Y	
MQCONN	Y	Y	
MQCONNX	Y		
MQDISC	Y	Y	
MQGET	Y	Y	
MQINQ	Y	Y	
MQOPEN	Y	Y	
MQPUT	Y	Y	
MQPUT1	Y	Y	
MQSET	Y	Y	
MQXCNVC	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 later sections in this book 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 whose size is not fixed. For these, a lower case "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 following table summarizes the fields in the structure.

Table 4. 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

Overview for MQAIR

Availability: AIX®, HP-UX, Solaris, Linux® and Windows® clients.

Purpose: The MQAIR structure allows an application running as a WebSphere 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 MQCONNX call.

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, respectively.

Fields for MQAIR

The MQAIR structure contains the following fields; the fields are described in **alphabetic 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. 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.

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 MQRC_AUTH_INFO_TYPE_ERROR.

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, *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.

AILUL (10-digit signed integer)

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.

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 *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.

  Using *LDAPUserNamePtr* is recommended 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).

  Using *LDAPUserNameOffset* is recommended 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 datatype, 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 and RPG declaration

Table 5. Initial values of fields in MQAIR for MQAIR

Field name	Name of constant	Value of constant
AISID	AISIDV	'AIRb'
AIVER	AIVERC	1
AITYP	AITLDP	1
AICN	None	Null string or blanks
AILUP	None	Null pointer or null bytes
AILUO	None	0
AILUL	None	0
AIPW	None	Null string or blanks
Notes: 1. The symbol b represents a single blank character.		

RPG declaration (copy file CMQAIRG)

```
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

The following table summarizes the fields in the structure.

Table 6. Fields in MQBMHO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	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 **alphabetic order**:

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.

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.

Initial values and RPG declaration

Buffer to message handle structure - Initial values

Table 7. 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 (copy file CMQBMHOG)

```
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 following table summarizes the fields in the structure.

Table 8. Fields in MQBO

Field	Description	Topic
BOSID	Structure identifier	BOSID
BOVER	Structure version number	BOVER
BOOPT	Options that control the action of MQBEGIN	BOOPT

Overview

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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively.

Fields

The MQBO structure contains the following fields; the fields are described in **alphabetic 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 and RPG declaration

Table 9. Initial values of fields in MQBO

Field name	Name of constant	Value of constant
BOSID	BOSIDV	'BObb'
BOVER	BOVER1	1
BOOPT	BONONE	0
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQBOG)

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQBO Structure
D*
D* Structure identifier
D  BOSID                  1      4    INZ('BO  ')
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.

The following table summarizes the fields in the structure.

Table 10. Fields in MQCBC

Field	Description	Topic
StrucID	Structure identifier	StrucID
Version	Structure version number	Version
CallType	Why function has been called	CallType
Hobj	Object handle	Hobj
CallbackArea	Field for callback function to use	CallbackArea
ConnectionArea	Field for callback function to use	ConnectionArea
CompCode	Completion code	CompCode
Reason	Reason code	Reason
State	Indication of the state of the current consumer	State
DataLength	Message length	DataLength
BufferLength	Length of message buffer in bytes	BufferLength
Flags	General flags	Flags

Overview for MQCBC

Availability: AIX, HP-UX, i5/OS, Solaris, Linux, Windows, z/OS®, plus WebSphere MQ 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_1.

Character set and encoding: Data in MQCBC will 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. However, if the application is running as an MQ 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:

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 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 *CBCLEN* 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 MQCC_WARNING, the value of the *Reason* field is MQRC_TRUNCATED_MSG_ACCEPTED 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, 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 *CBCCC* 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 CBDOPT in the MQCBD structure.

The *CBCLEN* 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 is 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 *CBCREA* field is MQRC_NONE.

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 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 *CBCREA* field is MQRC_NONE.

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 MQOP_STOP.

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 The exception handler function has been invoked without a message when: 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 *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 MQOP_STOP, 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*= MQCC_FAILED) has been detected that is specific to the object handle; for example *CBCREA* code = MQRC_GET_INHIBITED.

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:

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.

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.

CBCLEN (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. Callback context structure - DataLength field

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 GMRL 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 CBDMML in the MQCBD 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 theCBDMML in the MQCBD 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.

CBCFLG (10-digit signed integer)

Flags containing information about this consumer. Callback context structure - Flags field

The following option is defined:

CBCFBE

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.

CBCHOBJ (10-digit signed integer)

Callback context structure - Hobj 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 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

CBCREA (10-digit signed integer)

Callback context structure - Reason field

This is the reason code qualifying the *CompCode*

This is an input field. The initial value of this field is MQRC_NONE.

CBCSTATE (10-digit signed integer)

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.
Callback context structure - State field

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
CSNONE This reason code represents a normal call with no additional reason information	None; this is the normal operation.	0
CSSUST 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
CSSUSU 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
CSSUS 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
CSSTOP 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

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 and RPG declaration

Callback context structure - Initial values

Table 11. Initial values of fields in MQCBC

Field name	Name of constant	Value of constant
CBCSID	CBCSI	'CBCb'
CBCVER	CBCV1	1
CBCCALLT	None	0
CBCHOBJ	MQHO_UNUSABLE_HOBJ	-1
CBCCALLBA	None	Null pointer or null bytes
CBCCONNAREA	None	Null pointer or null bytes
CBCCC	MQCC_OK	0
CBCREA	MQRC_NONE	0
CBCSTATE	MQCS_NONE	0
CBCLEN	None	0
CBCBUFFLEN	None	0
Flags	None	0
Notes: 1. The symbol b represents a single blank character.		

RPG declaration (copy file CMQCBCG)

```
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  CBCLEN                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)
```

MQCBD – Callback descriptor

Structure specifying the callback function.

The following table summarizes the fields in the structure.

Table 12. Fields in MQCBD

Field	Description	Topic
CBDSID	Structure identifier	"CBDSID (10-digit signed integer)" on page 31
CBDVER	Structure version number	"CBDVER (10-digit signed integer)" on page 31
CBDCALLBT	Type of callback function	"CBDCALLBT (10-digit signed integer)" on page 29
CBDOPT	Options controlling message consumption	"CBDOPT (10-digit signed integer)" on page 30
CBDCALLBA	Field for callback function to use	"CBDCALLBA (10-digit signed integer)" on page 27
CBDCALLBF	Whether the function is invoked as an API call	"CBDCALLBF (10-digit signed integer)" on page 27
CBDCALLBN	Whether the function is invoked as a dynamically-linked program	"CBDCALLBN (10-digit signed integer)" on page 27
CBDMML	Length of longest message that can be read	"CBDMML (10-digit signed integer)" on page 29

Overview for MQCBD

Availability: AIX, HP-UX, i5/OS, Solaris, Linux, Windows, z/OS, and WebSphere MQ 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 and encoding of the local queue manager; these are given by the *CodedCharSetId*

queue-manager attribute and MQENC_NATIVE, respectively. However, if the application is running as an MQ 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:

CBDCALLBA (10-digit signed integer)

Callback descriptor structure - CBDCALLBA 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 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.

CBDCALLBF (10-digit signed integer)

Callback descriptor structure - CBDCALLBF field

The callback function is invoked as a function call.

Use this field to specify a pointer to the call back 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 not set, the call fails with the reason code MQRC_CALLBACK_ROUTINE_ERROR.

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 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.

CBDCALLBN (10-digit signed integer)

Callback descriptor structure - CallbackName field

The call back 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 either *CallbackName* or *CallbackFunction* is not true, 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:

i5/OS 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 WebSphere MQ if the program is to use WebSphere 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.

CBDCALLBT (10-digit signed integer)

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.

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. 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 "CBCLEN (10-digit signed integer)" on page 22 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.

CBDOPT (10-digit signed integer)

Callback descriptor structure - Options field

Any one, or all, of the following can be specified. If more than one option is required the values 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).

Combinations that are not valid are noted; any other combinations are valid.

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.

See "CBCCALLT (10-digit signed integer)" on page 19 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.

CBDSID (10-digit signed integer)
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.

CBDVER (10-digit signed integer)
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 RPG declaration
Callback descriptor structure - Initial values

Table 13. Initial values of fields in MQCBD

Field name	Name of constant	Value of constant
StrucId	MQCBD_STRUC_ID	'CBDb'
Version	MQCBD_VERSION_1	1
CallBackType	MQCBT_MESSAGE_CONSUMER	1

Table 13. Initial values of fields in MQCBD (continued)

Field name	Name of constant	Value of constant
Options	MQCBDO_NONE	0
CallbackArea	None	Null pointer or null blanks
CallbackFunction	None	Null pointer or null blanks
CallbackName	None	Null string or blanks
MaxMsgLength	MQCBD_FULL_MSG_LENGTH	-1

Notes:

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 listed above. Use it in the following way to provide initial values for the fields in the structure:

   ```
   MQCBD MyCBD = {MQCBD_DEFAULT};
   ```

RPG declaration (copy file MQCBDG)

```
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

The following table summarizes the fields in the structure.

Field	Description	Topic
VCHRP	Pointer to the variable length string	VCHRP

Field	Description	Topic
VCHRO	Offset in bytes of the variable length string from the start of the structure that contains this MQCHARV structure	VCHRO
VCHRS	Size in bytes of the buffer addressed by the VCHRP or VCHRO field.	VCHRS
VCHRL	The length in bytes of the variable length string addressed by the VCHRP or VCHRO field.	VCHRL
VCHRC	The character set identifier of the variable length string addressed by the VCHRP or VCHRO field.	VCHRC

Overview

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 ENNAT and the character set of the VCHRC 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 VCHRC is one of these character sets, the encoding used is the same encoding as that of the other fields in the MQCHARV.

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.

Fields

The MQCHARV structure contains the following fields; the fields are described in **alphabetic 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 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 exactly 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:

VSNLT

If VSNLT is not specified, VCHRL bytes are included as part of the string. If null characters are present they do not delimit the string.

If VSNLT 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 VSNLT 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 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 (seven 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' would be two valid Unicode characters and would 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 and RPG declaration

Initial values of fields in MQCHARV

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 for MQCHARV

```
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 following table summarizes the fields in the structure.

Table 14. Fields in MQCIH

Field	Description	Topic
CISID	Structure identifier	CISID
CIVER	Structure version number	CIVER
CILEN	Length of MQCIH structure	CILEN
CIENC	Reserved	CIENC
CICSI	Reserved	CICSI
CIFMT	MQ format name of data that follows MQCIH	CIFMT
CIFLG	Flags	CIFLG
CIRET	Return code from bridge	CIRET
CICC	MQ completion code or CICS EIBRESP	CICC

Table 14. Fields in MQCIH (continued)

Field	Description	Topic
CIREA	MQ reason or feedback code, or CICS EIBRESP2	CIREA
CIUOW	Unit-of-work control	CIUOW
CIGWI	Wait interval for MQGET call issued by bridge task	CIGWI
CILT	Link type	CILT
CIODL	Output COMMAREA data length	CIODL
CIFKT	Bridge facility release time	CIFKT
CIADS	Send/receive ADS descriptor	CIADS
CICT	Whether task can be conversational	CICT
CITES	Status at end of task	CITES
CIFAC	Bridge facility token	CIFAC
CIFNC	MQ call name or CICS EIBFN function	CIFNC
CIAC	Abend code	CIAC
CIAUT	Password or passticket	CIAUT
CIRS1	Reserved	CIRS1
CIRFM	MQ format name of reply message	CIRFM
CIRSI	Reserved	CIRSI
CIRTI	Reserved	CIRTI
CITI	Transaction to attach	CITI
CIFL	Terminal emulated attributes	CIFL
CIAI	AID key	CIAI
CISC	Transaction start code	CISC
CICNC	Abend transaction code	CICNC
CINTI	Next transaction to attach	CINTI
CIRS2	Reserved	CIRS2
CIRS3	Reserved	CIRS3
Note: The remaining fields are not present if *CIVER* is less than CIVER2.		
CICP	Cursor position	CICP
CIEO	Offset of error in message	CIEO
CIII	Reserved	CIII
CIRS4	Reserved	CIRS4

Overview

Purpose: The MQCIH structure describes the information that can be present at the start of a message sent to the CICS bridge through WebSphere MQ for z/OS.

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 16 on page 48, 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 15 shows which fields are set for different values of *CIRET*.

Table 15. Contents of error information fields in MQCIH structure

CIRET	*CIFNC*	*CICC*	*CIREA*	*CIAC*
CRC000	–	–	–	–
CRC003	–	–	FBC*	–
CRC002 CRC008	MQ call name	MQ *CMPCOD*	MQ *REASON*	–
CRC001 CRC006 CRC007 CRC009	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	–
CRC004 CRC005	–	–	–	CICS ABCODE

Fields

The MQCIH structure contains the following fields; the fields are described in **alphabetic 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.

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 justified.

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)

MQ completion code or CICS EIBRESP.

The value returned in this field is dependent on *CIRET*; see Table 15 on page 37.

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. Subsequently, 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 LNFAC. 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 LNFACL. 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)

MQ format name of data that follows MQCIH.

This specifies the 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)

MQ call name or CICS EIBFN function.

The value returned in this field is dependent on *CIRET*; see Table 15 on page 37. The following values are possible when *CIFNC* contains an 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 of time specified when the bridge was started.

WIULIM
Unlimited wait interval.

This is a request field. The initial value of this field is WIDFLT.

CIII (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 (usually 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 may 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)

MQ reason or feedback code, or CICS EIBRESP2.

The value returned in this field is dependent on *CIRET*; see Table 15 on page 37.

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 15 on page 37). 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') 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)

MQ format name of reply message.

This is the 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 whose initial program is 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 and RPG declaration

Table 16. Initial values of fields in MQCIH

Field name	Name of constant	Value of constant
CISID	CISIDV	'CIHb'
CIVER	CIVER2	2
CILEN	CILEN2	180
CIENC	None	0
CICSI	None	0
CIFMT	FMNONE	Blanks
CIFLG	CIFNON	0
CIRET	CRC000	0
CICC	CCOK	0
CIREA	RCNONE	0
CIUOW	CUONLY	273
CIGWI	WIDFLT	-2
CILT	LTPROG	1
CIODL	OLINPT	-1
CIFKT	None	0
CIADS	ADNONE	0
CICT	CTNO	0
CITES	TENOSY	0
CIFAC	FCNONE	Nulls
CIFNC	CFNONE	Blanks
CIAC	None	Blanks
CIAUT	None	Blanks
CIRS1	None	Blanks
CIRFM	FMNONE	Blanks
CIRSI	None	Blanks
CIRTI	None	Blanks
CITI	None	Blanks
CIFL	None	Blanks
CIAI	None	Blanks
CISC	SCNONE	Blanks
CICNC	None	Blanks
CINTI	None	Blanks

Table 16. Initial values of fields in MQCIH (continued)

Field name	Name of constant	Value of constant
CIRS2	None	Blanks
CIRS3	None	Blanks
CICP	None	0
CIEO	None	0
CIII	None	0
CIRS4	None	0
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQCIHG)

```
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 options

The following table summarizes the fields in the structure.

Table 17. Fields in MQCMHO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options	Options

Overview for MQCMHO

Availability: AIX, HP-UX, i5/OS, Solaris, Linux, Windows, z/OS and WebSphere 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 **alphabetic 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 will be 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 will be 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.

CMDEFV

This specifies that the default level of validation of property names should occur.

The default level of validation is equivalent to that specified by MQCMHO_VALIDATE.

In a future release an administrative option may be defined which will change the level of validation that will occur when MQCMHO_DEFAULT_VALIDATION is defined.

This is the default value.

CMNOVA

No validation on the property name will occur. See the description of MQCMHO_VALIDATE.

Default option: If none of the options described above 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. MQCMHO_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 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 and RPG declaration

Table 18. 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 (copy file CMQCMHOG)

```
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 following table summarizes the fields in the structure.

Table 19. Fields in MQCNO

Field	Description	Topic
CNSID	Structure identifier	CNSID
CNVER	Structure version number	CNVER
CNOPT	Options that control the action of MQCONNX	CNOPT
Note: The remaining fields are ignored if *CNVER* is less than CNVER2.		
CNCCO	Offset of MQCD structure for client connection	CNCCO
CNCCP	Address of MQCD structure for client connection	CNCCP
Note: The remaining fields are ignored if *CNVER* is less than CNVER3.		
CNCT	Queue-manager connection tag	CNCT
Note: The remaining fields are ignored if *CNVER* is less than CNVER4.		
CNSCP	Address of MQSCO structure for client connection	CNSCP
CNSCO	Offset of MQSCO structure for client connection	CNSCO
Note: The remaining fields are ignored if *CNVER* is less than CNVER5.		
CNCONID	Connection ID (a unique connection identifier)	CNCONID
CNSECPO	Security parameters offset	CNSECPO
CNSECPP	Security parameters pointer	CNSECPP

Overview

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 MQCONNX call.

Version: The current version of MQCNO is CNVER4. 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively.

Fields

The MQCNO structure contains the following fields; the fields are described in **alphabetic 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.

You can use CNCCO only when the application issuing the MQCONNX call is running as a WebSphere MQ client. It is therefore not applicable to the i5/OS platform.

CNCCP (pointer)

This is a pointer to an MQCD channel definition structure

You can use CNCCP only when the application issuing the MQCONNX call is running as a WebSphere MQ client. It is therefore not applicable to the i5/OS platform.

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 on 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 MQCONNX.

Binding options: The following options control the type of MQ binding that will be used; specify only one of these options:

CNSBND
Standard binding.

This option causes the application and the local-queue manager agent (the component that manages queuing operations) to run in separate units of execution (generally, in separate processes). This 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 may not have been fully tested, or may be unreliable or untrustworthy. CNSBND is the default.

CNSBND is defined to aid program documentation. It is not intended that this option be used with any other option controlling the type of binding used, but as its value is zero, such use cannot be detected.

This option is supported in all environments.

CNFBND
Fastpath binding.

This option causes the application and the local-queue manager agent to be part of the same unit of execution. This is in contrast to the normal method of 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 may be of advantage in situations where the use of multiple processes is a significant performance overhead compared to the overall resource used by the application. An application that uses the fastpath binding is known as a *trusted application*.

The following important points must be considered when deciding whether to use the fastpath binding:

- **Use of the CNFBND option compromises the integrity of the queue manager, because it permits a rogue application to alter or corrupt messages and other data areas belonging to the queue manager. It should therefore be considered for use *only* in situations where these issues have been fully evaluated.**
- 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. Refer to the WebSphere MQ Application Programming Guide for more information.
- 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 i5/OS, the job must run under user profile QMQM that belongs to the QMQMADM group. Also, the program must not terminate abnormally, otherwise unpredictable results may occur.

For more information about the implications of using trusted applications, see the WebSphere MQ Application Programming Guide.

MQCNO_SHARED_BINDING
Shared Bindings.

This option causes the application and the local-queue-manager agent (the component that manages queuing operations) to run in separate units of execution (generally, in separate processes). This 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. 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
Isolated Bindings.

This option causes the application and the local-queue-manager agent (the component that manages queuing operations) to run in separate units of execution (generally, in separate processes). This 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. 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.

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.

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, 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.

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.

CNHSNB
Serial handle sharing between threads, without call blocking.

This is the same as CNHSB, 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 handle, 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 the second and later 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 share properties as the connection handle specified on the MQOPEN call that created the object handle. Also, units of work inherit the same share 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 no handle-sharing option is specified, 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.

Default option: If none of the options described above is required, the following option can be used:

CNNONE
No options specified.

CNNONE is defined to aid program documentation. It is not intended that this option be used with any other CN* option, but as its value is zero, such use cannot be detected.

This is always an input field. The initial value of this field is CNNONE.

CNSCO (10-digit signed integer)

This is the offset in bytes of an MQSCO structure from the start of the MQCNO structure.

You can use CNSCP only when the application issuing the MQCONNX call is running as a WebSphere MQ client. It is therefore not applicable on the i5/OS platform.

CNSCP (pointer)

This is the address of an MQSCO structure.

You can use CNSCP only when the application issuing the MQCONNX call is running as a WebSphere MQ client. It is therefore not applicable on the i5/OS platform.

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.

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.

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 and RPG declaration

Table 20. Initial values of fields in MQCNO

Field name	Name of constant	Value of constant	
CNSID	CNSIDV	'CNOb'	
CNVER	CNVER1	1	
CNOPT	CNNONE	0	
CNCCO	None	0	
CNCCP	None	Null pointer or null bytes	
CNCT	CTNONE	Nulls	
CNSCP	None	Null pointer or null bytes	
CNSCO	None	0	
CNCONID	None	Nulls	
CNSECPO	None	0	
CNSECPP	None	Null pointer or null bytes	
Notes: 1. The symbol 'b' represents a single blank character.			

RPG declaration (copy file CMQCNOG)

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D*
D* MQCNO Structure
D*
D* Structure identifier
D  CNSID                  1      4    INZ('CNO ')
D* Structure version number
D  CNVER                  5      8I 0 INZ(1)
D* Options that control the action ofMQCONNX
D  CNOPT                  9     12I 0 INZ(0)
D* Offset of MQCD structure for clientconnection
D  CNCCO                 13     16I 0 INZ(0)
D* Address of MQCD structure for clientconnection
D  CNCCP                 17     32*   INZ(*NULL)
D* Queue-manager connection tag
D  CNCT                  33    160    INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     0000000000000000000-
D                                     ')
D* Address of MQSCO structure forclient connection
D  CNSCP                 161    176*   INZ(*NULL)
D* Offset of MQSCO structure for clientconnection
D  CNSCO                 177    180I 0 INZ(0)
D* Unique Connection Identifier
D  CNCONID               181    204    INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D* Offset of MQCSP structure
D  CNSECPO               205    208I 0 INZ(0)
D* Address of MQCSP structure
D  CNSECPP               209    224*   INZ(*NULL)
```

MQCSP - Security parameters

Summary of the MQCSP structure for WebSphere MQ for i5/OS.

The following table summarizes the fields in the structure.

Table 21. Fields in MQCSP

Field	Description	Topic
CSSID	Structure identifier	CSSID
CSVER	Structure version number	CSVER
CSAUTHT	Type of authentication	CSAUTHT
CSRE1	Required for pointer alignment on i5/OS	CSRE1
CSCSPUIP	Address of user ID	CSCSPUIP
CSCSPUIO	Offset of user ID	CSCSPUIO
CSCSPUIL	Length of user ID	CSCSPUIL

Table 21. Fields in MQCSP (continued)

Field	Description	Topic
CSRS2	Required for pointer alignment on i5/OS	CSRS2
CSCPPP	Address of password	CSCPPP
CSCPPO	Offset of password	CSCPPO
CSCPPL	Length of password	CSCPPL

Overview for MQCSP

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 MQCONNX call.

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 ENAT, respectively.

Fields for MQCSP

The MQCSP structure contains the following fields; the fields are described in **alphabetic order**.

CSAUTHT (10-digit signed integer)

This is the type of authentication to perform.

Valid values are:

CSAN
Do not use user ID and password fields.

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 MQRC_NOT_AUTHORIZED.

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 MQRC_NOT_AUTHORIZED.

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 i5/OS.

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 i5/OS.

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 and RPG declaration

Table 22. Initial values of fields in MQCNO

Field name	Name of constant	Value of constant	
CSSID	CSSIDV	'CSPb'	
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	
CSCPPO	None	0	
CSCPPL	None	0	
Note: 1. The symbol 'b' represents a single blank character.			

Initial values and RPG declaration

RPG declaration (copy file CMQCSPG)

```
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  CSCPPO                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.

The following table summarizes the fields in the structure.

Table 23. Fields in MQCTLO

Field	Description	Topic
StrucID	Structure identifier	StrucID
Version	Structure version number	Version
Options	Options	Options
Reserved	Reserved field	Options
ConnectionArea	Field for callback function to use	ConnectionArea

Overview for MQCTLO

Availability: AIX, HP-UX, i5/OS, Solaris, Linux, Windows, z/OS, and WebSphere MQ 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively. However, if the application is running as an MQ 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:

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 ConnectionArea field in the MQCBC structure, which is a parameter on the MQCB call.

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.

COOPT (10-digit signed integer)

Control options structure - Options field

Options that control the action of MQCTLO.

MQCTLO_FAIL_IF_QUIESCING
Force the MQCTLO 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.

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 *COOPT* field is MQCTLO_NONE.

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 and RPG declaration
Control options structure - Initial values

Table 24. Initial values of fields in MQCTLO

Field name	Name of constant	Value of constant
COSID	CTLSI	'CTLO'

Table 24. Initial values of fields in MQCTLO (continued)

Field name	Name of constant	Value of constant
COVER	CTLV1	1
COOPT	MQCTLO_NONE	Nulls
CORSV	Reserved field	
COCONNAREA	None	Null pointer or null bytes

RPG declaration (copy file CTLOG)

```
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 following table summarizes the fields in the structure.

Table 25. Fields in MQDH

Field	Description	Topic
DHSID	Structure identifier	DHSID
DHVER	Structure version number	DHVER
DHLEN	Length of MQDH structure plus following records	DHLEN
DHENC	Numeric encoding of data that follows array of MQPMR records	DHENC
DHCSI	Character set identifier of data that follows array of MQPMR records	DHCSI
DHFMT	Format name of data that follows array of MQPMR records	DHFMT
DHFLG	General flags	DHFLG
DHPRF	Flags indicating which MQPMR fields are present	DHPRF
DHCNT	Number of object records present	DHCNT
DHORO	Offset of first object record from start of MQDH	DHORO
DHPRO	Offset of first put-message record from start of MQDH	DHPRO

Overview

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 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.

Format name: FMDH.

Character set and encoding: Data in MQDH must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT for the C programming language, respectively.

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 may 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 above, 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 437). 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

The MQDH structure contains the following fields; the fields are described in **alphabetic 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 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 be 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 and RPG declaration

Table 26. Initial values of fields in MQDH

Field name	Name of constant	Value of constant
DHSID	DHSIDV	'DHbb'
DHVER	DHVER1	1
DHLEN	None	0
DHENC	None	0
DHCSI	CSUNDF	0
DHFMT	FMNONE	Blanks
DHFLG	DHFNON	0
DHPRF	PFNONE	0
DHCNT	None	0
DHORO	None	0
DHPRO	None	0
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQDHG)

```
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

The following table summarizes the fields in the structure.

Table 27. Fields in MQDLH

Field	Description	Topic
DLSID	Structure identifier	DLSID
DLVER	Structure version number	DLVER
DLREA	Reason message arrived on dead-letter queue	DLREA
DLDQ	Name of original destination queue	DLDQ
DLDM	Name of original destination queue manager	DLDM
DLENC	Numeric encoding of data that follows MQDLH	DLENC
DLCSI	Character set identifier of data that follows MQDLH	DLCSI
DLFMT	Format name of data that follows MQDLH	DLFMT
DLPAT	Type of application that put message on dead-letter queue	DLPAT
DLPAN	Name of application that put message on dead-letter queue	DLPAN
DLPD	Date when message was put on dead-letter queue	DLPD
DLPT	Time when message was put on dead-letter queue	DLPT

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 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: FMDLH.

Character set and encoding: The fields in the MQDLH structure are in the character set and encoding given by the *MDCSI* and *MDENC* fields in the header structure that precedes MQDLH, or by those fields 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.

Usage: Applications that put messages directly on the dead-letter queue should 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 should consider doing 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, this should be done 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 above is appropriate (if any) 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: When a message is put on the dead-letter queue, the MQMD structure used for the MQPUT or MQPUT1 call should be identical to the MQMD associated with the message (usually the MQMD returned by the MQGET call), with the exception of the following:

- 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 a MQDLH structure.
- The context fields (*MDACC*, *MDAID*, *MDAOD*, *MDPAN*, *MDPAT*, *MDPD*, *MDPT*, *MDUID*) should 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 should use the PMDEFC 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 it has just received should use the PMPASA option, in order to preserve the original context information.

– A server application putting on the dead-letter queue a *reply* to a message it has just received should use the PMPASI 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 it received from its communication channel should use the PMSETA option, to preserve the original context information.

In the MQDLH structure itself, the fields should be set as follows:

- The *DLCSI*, *DLENC* and *DLFMT* fields should be set to the values that describe the data that follows the MQDLH structure, usually the values from the original message descriptor.
- The context fields *DLPAT*, *DLPAN*, *DLPD*, and *DLPT* should 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 should be set as appropriate.

The application should 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 should not be terminated 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 should 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. Applications that get messages from the dead-letter queue should also be aware that such messages may have been truncated if they were originally too long for the queue.

Fields

The MQDLH structure contains the following fields; the fields are described in **alphabetic order**:

DLCSI (10-digit signed integer)

Character set identifier of data that follows MQDLH.

This specifies the character set identifier of the data that follows 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:

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 LNQMN. 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.

This specifies 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.

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 (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 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, also, the description of the *MDPAN* field in "MQMD – Message descriptor" on page 125.

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, 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 125 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 8 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 below)

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 *DLPT*. This happens 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 8 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 should be one of the FB* or RC* values (for example, RC2053). See the description of the *MDFB* field in "MQMD – Message descriptor" on page 125 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 have been transferred to the dead-letter queue. These 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 274).

FBATYP
Application type error.

An application processing a trigger message was unable to start the application because the *TMAT* field of the trigger message is not valid (see "MQTM – Trigger message" on page 274).

FBBOCD
Cluster-receiver channel deleted.

The message was on the SYSTEM.CLUSTER.TRANSMIT.QUEUE intended for a cluster queue that had been opened with the OOBNDO 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 OOBNDO 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 has been 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 may not be valid, the *TMVER* may not be recognized, or the length of the trigger message may be insufficient to contain the MQTM structure.

FBXQME
Message on transmission queue not in correct format.

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.

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 and RPG declaration

Table 28. Initial values of fields in MQDLH

Field name	Name of constant	Value of constant
DLSID	DLSIDV	'DLHb'
DLVER	DLVER1	1
DLREA	RCNONE	0
DLDQ	None	Blanks
DLDM	None	Blanks
DLENC	None	0
DLCSI	CSUNDF	0
DLFMT	FMNONE	Blanks

Table 28. Initial values of fields in MQDLH (continued)

Field name	Name of constant	Value of constant
DLPAT	None	0
DLPAN	None	Blanks
DLPD	None	Blanks
DLPT	None	Blanks
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQDLHG)

```
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 queuemanager
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 thatfollows 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 messageon dead-letter
D* (undelivered-message)queue
D  DLPAT                125    128I 0 INZ(0)
D* Name of application that put messageon dead-letter
D* (undelivered-message)queue
D  DLPAN                129    156    INZ
D* Date when message was put ondead-letter (undelivered-message)queue
D  DLPD                 157    164    INZ
D* Time when message was put on thedead-letter (undelivered-message)queue
D  DLPT                 165    172    INZ
```

MQDMHO – Delete message handle options

The following table summarizes the fields in the structure.

Table 29. Fields in MQDMHO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options	Options

Overview for MQDMHO

Availability: All WebSphere MQ systems and WebSphere 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 **alphabetic 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 and RPG declaration

Table 30. 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 (copy file MQDMHOG)

```
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

The following table summarizes the fields in the structure.

Table 31. Fields in MQDMPO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options controlling the action of MQDMPO	Options

Overview for MQDMPO

Availability: All WebSphere MQ systems and WebSphere 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 **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.

MQDMPO_DEL_NEXT
Deletes the next property that matches the specified name, continuing the search from the property cursor. If this is the first MQDLTMP call for the specified name, the first property that matches the specified name is deleted.

If the property under the cursor has been deleted, MQINQMP deletes the next matching property following the one that has been deleted.

If a property is added that matches the specified name while iteration is in progress, the property might be deleted during the completion of the iteration. The property will be deleted once the iteration is restarted with MQDMPO_DEL_FIRST.

DPDELC
Deletes the property pointed to by the property cursor; that is the property 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, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQGET or MQPUT call respectively.

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 already been deleted, the call fails with completion code MQCC_FAILED and reason MQRC_PROPERTY_NOT_AVAILABLE.

If none of the options described above is required, the following option can be used:

DPNONE
No options specified.

This is always an input field. The initial value of this 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.

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 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 is always an input field. The initial value of this field is DPVER1

Initial values and RPG declaration

Delete message property options structure - Initial values

Table 32. Initial values of fields in MQDPMO

Field name	Name of constant	Value of constant
DPSID	DPSIDV	'DMPO'
DPVER	DPVER1	1
DPOPT	Options that control the action of MQDLTMP	MQDMPO_NONE

RPG declaration (copy file MQDMPOG)

```
D* MQDMPO Structure
D*
D*
D* Structure identifier
D  DPSID                  1      4    INZ('DMPO')
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

The following table summarizes the fields in the structure.

Table 33. Fields in MQEPH

Field	Description	Topic
EPSID	Structure identifier	EPSID
EPVER	Structure version number	EPVER
EPLEN	Length of MQEPH structure plus the MQCFH and parameter structures that follow it	EPLEN
EPENC	Numeric encoding of data that follows last PCF parameter structure	EPENC
EPCSI	Character set identifier of data that follows last PCF parameter structure	EPCSI
EPFMT	Format name of data that follows last PCF parameter structure	EPFMT
EPFLG	Flags	EPFLG
EPPCFH	Programmable command format (PCF) header	EPPFH

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

The MQEPH structure contains the following fields; the fields are described in **alphabetic 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 is 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 34. Initial values of fields in EPPCFH

Field name	Name of constant	Value of constant
EP3TYP	MQCFT_NONE	0
EP3LEN	MQCFH_STRUC_LENGH	36
EP3VER	MQCFH_VERSION_3	3
EP3CMD	MQCMD_NONE	0
EP3SEQ	None	1
EP3CTL	MQCFC_LAST	1
EEP3CC	MQCC_OK	0
EP3REA	MQRC_NONE	0
EP3CNT	None	0

The application must change EP3TYP from MQCFT_NONE 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 and language declarations

Table 35. Initial values of fields in MQEPH

Field name	Name of constant	Value of constant
EPSID	EPSTID	'EPbb'
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 34 on page 84	0
Notes: 1. The symbol b represents a single blank character.		

RPG declaration (copy file CMQEPHG)

```
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 lenght 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 following table summarizes the fields in the structure.

Table 36. Fields in MQGMO

Field	Description	Topic
GMSID	Structure identifier	GMSID
GMVER	Structure version number	GMVER
GMOPT	Options that control the action of MQGET	GMOPT
GMWI	Wait interval	GMWI
GMSG1	Signal	GMSG1
GMSG2	Signal identifier	GMSG2
GMRQN	Resolved name of destination queue	GMRQN
Note: The remaining fields are ignored if *GMVER* is less than GMVER2.		
GMMO	Options controlling selection criteria used for MQGET	GMMO
GMGST	Flag indicating whether message retrieved is in a group	GMGST
GMSST	Flag indicating whether message retrieved is a segment of a logical message	GMSST
GMSEG	Flag indicating whether further segmentation is allowed for the message retrieved	GMSEG
GMRE1	Reserved	GMRE1
Note: The remaining fields are ignored if *GMVER* is less than GMVER3.		
GMTOK	Message token	GMTOK
GMRL	Length of message data returned (bytes)	GMRL

Overview

Purpose: The MQGMO structure allows the application to specify options that 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 GMVER3. 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue

manager attribute and ENNAT, respectively. However, if the application is running as an MQ client, the structure must be in the character set and encoding of the client.

Fields

The MQGMO structure contains the following fields; the fields are described in **alphabetic 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 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 GSNIG. This field is ignored if *GMVER* is less than GMVER2.

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 will be 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 match options can be used to select which of the eligible messages is the one actually 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 of the above 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 is this condition is not satisfied.

GMMO is ignored if either GMMUC or GMBRWC is specified.

One or more of the following match 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 may 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 will match.

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 may 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 will match.

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 may 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 will match.

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 may 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 will match.

MOOFFS
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 may 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 will match.

If none of the options described above 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 be used with any other MO* option, but as its value is zero, such use cannot be detected.

This 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 verison 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 prior to 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 options described below can be specified. If more than one is required the values can be added together (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 *GMWI*.

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 below.

Note: In the description below, 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 of 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 should 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 may 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 neither this option nor GMNSYP is 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 (see above for details).
- If the message is not persistent, the queue manager processes the call as though the application had specified GMNSYP (see below 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 neither this option nor GMSYP is 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 OOBRW 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 will search 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 subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that 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.

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 prior to the call.

This option is not valid with any of the following options:

- GMBRWC
- GMBRWN
- GMMUC
- GMSYP
- GMPSYP
- 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 437 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 will not be 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 subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that 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 subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that 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 may 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 whose *MDOFF* field in MQMD 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 whose *MDOFF* field in MQMD 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 of the message segments that comprise the logical message are locked to the queue handle (provided that 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 OOBRW, 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*, *BUFLEN*, *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 OOBRW 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 should be 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 125 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 may 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 extremely 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 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 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 will be 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 may 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 may 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 prior to each MQGET call. Specifically, it means that the application does not need to set the group and segment information prior to 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 "MQGMO – Get-message options" on page 86 for further details.

Table 37 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 prior to the call.

Cur log msg
Indicates whether a current logical message exists prior to 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 37. MQGET options relating to messages in groups and segments of logical messages

Options you specify	Group and log-msg status prior to call		Values the queue manager looks for				
LOG ORD	**Cur grp**	**Cur log msg**	*MDMID*	*MDCID*	*MDGID*	*MDSEQ*	*MDOFF*
Yes	No	No	Controlled by *GMMO*	Controlled by *GMMO*	Controlled by *GMMO*	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 *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 of 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 may be returned out of order, or they may 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 86 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 desired, 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 should 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 may however succeed with a CCWARN completion code. Table 38 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).

Table 38. Outcome when MQGET or MQCLOSE call is not consistent with group and segment information

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 may 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, prior to 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 may arrive out of order. By specifying neither GMLOGO, nor 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 (see above).

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 wish to receive individual segments should therefore always specify GMCMPM.

To use this option, the application must provide a buffer which is big 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 may 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 partway 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 will be the same for all segments in the logical message – usually 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 may 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 may 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* and/or *MDENC* 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 of 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 whose *MDOFF* field in MQMD 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 GMPSYP, 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 may 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 has an effect *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 of 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 may 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 comprise 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 may 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 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.

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 above is 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 be 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 usually given by the smaller of *BUFLEN* 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 will be 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.

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 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 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.

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 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 125 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 in conjunction 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 and RPG declaration

Table 39. Initial values of fields in MQGMO

Field name	Name of constant	Value of constant
GMSID	GMSIDV	'GMOb'
GMVER	GMVER1	1
GMOPT	GMNWT	0
GMWI	None	0
GMSG1	None	0
GMSG2	None	0
GMRQN	None	Blanks
GMMO	MOMSGI + MOCORI	3
GMGST	GSNIG	'b'
GMSST	SSNSEG	'b'
GMSEG	SEGIHB	'b'
GMRE1	None	'b'
GMTOK	MTKNON	Nulls
GMRL	RLUNDF	-1

Notes:
1. The symbol 'b' represents a single blank character.

RPG declaration (copy file CMQGMOG)

```
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 ofMQGET
D  GMOPT                  9     12I 0 INZ(0)
```

```
D* Wait interval
D  GMWI                  13     16I 0 INZ(0)
D* Signal
D  GMSG1                 17     20I 0 INZ(0)
D* Signal identifier
D  GMSG2                 21     24I 0 INZ(0)
D* Resolved name of destination queue
D  GMRQN                 25     72    INZ
D* Options controlling selection criteriaused for MQGET
D  GMMO                  73     76I 0 INZ(3)
D* Flag indicating whether messageretrieved is in a group
D  GMGST                 77     77    INZ(' ')
D* Flag indicating whether messageretrieved is a segment of a
D* logicalmessage
D  GMSST                 78     78    INZ(' ')
D* Flag indicating whether furthersegmentation is allowed for themessage
D* retrieved
D  GMSEG                 79     79    INZ(' ')
D* Reserved
D  GMRE1                 80     80    INZ
D* Message token
D  GMTOK                 81     96    INZ(X'00000000000000-
D                                     0000000000000000')
D* Length of message data returned(bytes)
D  GMRL                  97    100I 0 INZ(-1)
```

MQIIH – IMS information header

The following table summarizes the fields in the structure.

Table 40. Fields in MQIIH

Field	Description	Topic
IISID	Structure identifier	IISID
IIVER	Structure version number	IIVER
IILEN	Length of MQIIH structure	IILEN
IIENC	Reserved	IIENC
IICSI	Reserved	IICSI
IIFMT	MQ format name of data that follows MQIIH	IIFMT
IIFLG	Flags	IIFLG
IILTO	Logical terminal override	IILTO
IIMMN	Message format services map name	IIMMN
IIRFM	MQ format name of reply message	IIRFM
IIAUT	RACF™ password or passticket	IIAUT
IITID	Transaction instance identifier	IITID
IITST	Transaction state	IITST
IICMT	Commit mode	IICMT
IISEC	Security scope	IISEC
IIRSV	Reserved	IIRSV

Overview

Purpose: The MQIIH structure describes the information that must be present at the start of a message sent to the IMS bridge through WebSphere MQ for z/OS.

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: It is now possible for WebSphere 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 WebSphere MQ, that is, no application name flows on the authentication request, and the MVSxxxx value to is used instead.

A value of between 1 and 8 alphanumeric characters must follow the rules for passticket application names as described in the RACF® publications.

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 userids of all applications that are to be granted access. The WebSphere 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 **alphabetic 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)

MQ format name of data that follows MQIIH.

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 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)

MQ format name of reply message.

This is the 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 desired 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 neither ISSCHK nor ISSFUL is 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 the WebSphere MQ Application Programming Guide 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 and RPG declaration

Table 41. Initial values of fields in MQIIH

Field name	Name of constant	Value of constant
IISID	IISIDV	'IIHb'
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	'b'
IICMT	ICMCTS	'0'
IISEC	ISSCHK	'C'

Table 41. Initial values of fields in MQIIH (continued)

Field name	Name of constant	Value of constant
IIRSV	None	'ƀ'
Notes: 1. The symbol 'ƀ' represents a single blank character.		

RPG declaration (copy file CMQIIHG)

```
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'00000000000000-
D                                     0000000000000000')
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 following table summarizes the fields in the structure. MQIMPO structure - inquire message property options

Table 42. Fields in MQIMPO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options controlling the action of MQINQMP	Options
RequestedEncoding	Encoding into which the enquired property is to be converted	RequestedEncoding

Table 42. Fields in MQIMPO (continued)

Field	Description	Topic
RequestedCCSID	Character set of the inquired property	RequestedCCSID
ReturnedEncoding	Encoding of the returned value	ReturnedEncoding
ReturnedCCSID	Character set of returned value	ReturnedCCSID
Reserved1	Reserved field	Reserved1
ReturnedName	Name of the inquired property	ReturnedName
TypeString	String representation of the data type of the property	TypeString

Overview for MQIMPO

The inquire message properties options structure.

Availability: All WebSphere MQ systems and WebSphere 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 **alphabetic order**:

IPOPT (10-digit signed integer)

Inquire message property options structure - IPOPT field

The following options control the action of MQINQMP. You can specify one or more of these options, and if you need more than one, the values 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).

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 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 *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 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.

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 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, WebSphere MQ returns reason code MQRC_PROP_NUMBER_FORMAT_ERROR.
- If an application attempts a conversion that is not supported, WebSphere 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; WebSphere 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.

  WebSphere 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.

WebSphere 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″.

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 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

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 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

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 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:

IPNONE

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 IPINQF.

IPREQCSI (10-digit signed integer)

Inquire message property options structure - IPREQCSI 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.

IPREQENC (10-digit signed integer)

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.

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)

Inquire message property options structure - IPRETCSI 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.

IPRETENC (10-digit signed integer)

Inquire message property options structure - IPRETENC 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.

IPRETNAMCHRP (10-digit signed integer)

Inquire message property options structure - IPRETNAMCHRP 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.

IPSID (10-digit signed integer)

Inquire message property options structure - IPSID field

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)

Inquire message property options structure - IPTYP 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.

IPVER (10-digit signed integer)

Inquire message property options structure - Version field

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 and RPG declaration

Inquire message property options structure - Initial values

Table 43. Initial values of fields in MQIPMO

Field name	Name of constant	Value of constant
IPSID	IPSIDV	'IMPO'
IPVER	IPVER1	1
IPOPT	IPINQF	
IPREQENC	MQENC_NATIVE	
IPREQCSI	MQCCSI_APPL	
IPRETENC	MQENC_NATIVE	
IPRETCSI	0	
IPRE1	0	
IPRETNAMCHRP	MQCHARV_DEFAULT	
IPTYP	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 listed above. Use it in the following way to provide initial values for the fields in the structure:

```
MQIMPO MyIMPO = {MQIMPO_DEFAULT};
```

RPG declaration (copy file MQIMPOG)

```
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  IPRETNAMCHRP          33     48*   INZ(*NULL)
D* Offset of variable length string
D  IPRETNAMCHRO          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

The following table summarizes the fields in the structure.

Table 44. Fields in MQMD

Field	Description	Topic
MDSID	Structure identifier	MDSID
MDVER	Structure version number	MDVER
MDREP	Options for report messages	MDREP
MDMT	Message type	MDMT
MDEXP	Message lifetime	MDEXP
MDFB	Feedback or reason code	MDFB
MDENC	Numeric encoding of message data	MDENC

Table 44. Fields in MQMD (continued)

Field	Description	Topic
MDCSI	Character set identifier of message data	MDCSI
MDFMT	Format name of message data	MDFMT
MDPRI	Message priority	MDPRI
MDPER	Message persistence	MDPER
MDMID	Message identifier	MDMID
MDCID	Correlation identifier	MDCID
MDBOC	Backout counter	MDBOC
MDRQ	Name of reply queue	MDRQ
MDRM	Name of reply queue manager	MDRM
MDUID	User identifier	MDUID
MDACC	Accounting token	MDACC
MDAID	Application data relating to identity	MDAID
MDPAT	Type of application that put the message	MDPAT
MDPAN	Name of application that put the message	MDPAN
MDPD	Date when message was put	MDPD
MDPT	Time when message was put	MDPT
MDAOD	Application data relating to origin	MDAOD
Note: The remaining fields are ignored if *MDVER* is less than MDVER2.		
MDGID	Group identifier	MDGID
MDSEQ	Sequence number of logical message within group	MDSEQ
MDOFF	Offset of data in physical message from start of logical message	MDOFF
MDMFL	Message flags	MDMFL
MDOLN	Length of original message	MDOLN

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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. However, if the application is running as an MQ 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 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 below.

- 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 48 on page 183.

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 286 for details.

Message context: Certain fields in MQMD contain the message context. Usually:

- **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 above, 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).

In the descriptions below, the context fields are described as though they are used as described above. 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 WebSphere 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 **alphabetic 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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 i5/OS, the accounting information is set to the accounting code for the job.

- On HP OpenVMS, Compaq NonStop Kernel, and UNIX systems, the accounting information is set to the numeric user identifier, in ASCII characters.
- On OS/2®, the accounting information is set to the ASCII character '1'.
- 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.

  ATTOS2
  OS/2 default accounting token.

  ATT400
  i5/OS accounting token.

  ATTUNX
  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, OS/2, i5/OS, Solaris, Windows, plus WebSphere MQ 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 the WebSphere MQ Application Programming Guide.

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 "PMOPT (10-digit signed integer)" on page 204 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 LNACCT. 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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 the WebSphere MQ Application Programming Guide.

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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 437.

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 PMOPT 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)

Character set identifier of message data.

This specifies the character set identifier of character data in the message.

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 471 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 whose identifier 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 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 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) whilst 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 whose expiry time 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 below (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 wish 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
Userid 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 WebSphere 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.

FBCCSE
Character set identifier not valid.

FBCIHE
CICS information header structure missing or not valid.

FBCCAE
Length of CICS commarea not valid.

FBCCIE
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 above.

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. Refer to the WebSphere MQ Programmable Command Formats and Administration Interface book for more information about using programmable command format messages.

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 71 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 65 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; Refer to the WebSphere MQ Programmable Command Formats and Administration Interface book for more information about this structure, and to the Monitoring WebSphere MQ book for information about events.

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 `llzzccc`, where:

ll is a 2-byte length field specifying the total length of the IMS variable string item. This length is equal to the length of `ll` (2 bytes), plus the length of `zz` (2 bytes), plus the length of the character string itself. `ll` 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 `ll-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 178 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. Refer to the WebSphere MQ Programmable Command Formats and

Administration Interface book 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 234 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 274 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 286 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.

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 *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. This is the recommended 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 Table 56 on page 208. 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 37 on page 100. 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 86 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 prior to *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 (see below), 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 (see below). 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 above rules, but otherwise splits messages as it thinks fit; no assumptions should be made about the way that the queue manager will choose to split a particular message.

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:

MFMIG
Message is a member of a group.

MFLMIG
Message is the last logical message in a group.

If this flag is set, the queue manager turns on MFMIG 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, MFLMIG 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 56 on page 208 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 Chapter 8, "Report options and message flags," on page 517.

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 [1] 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

1. An *MDMID* generated by the queue manager consists of a 4-byte product identifier ('AMQb' or 'CSQb' in either ASCII or EBCDIC, where 'b' represents a blank), followed by a product-specific implementation of a unique string. In WebSphere 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.

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 the message is retained then the message identifier returned on the MQPUT is the message identifier of the retained message, otherwise this message identifier will not represent any message. The value of the *MDMID* field in the MQMD is unchanged on return from the call if MINONE or PMNMID was specified.

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 above
- 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 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 37 on page 100 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 LNMID. 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 above.

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 56 on page 208. 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 37 on page 100. 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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 WebSphere MQ for OS/390®, the 8-character jobname of the address space created for the OpenEdition™ 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 OS/2, 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 i5/OS, the queue manager uses the fully-qualified job name.
- On HP OpenVMS and Compaq NonStop Kernel, 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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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
WebSphere MQ 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.

ATOS2
OS/2 or Presentation Manager application.

AT390 OS/390 application (same value as ATZOS).

AT400 i5/OS application.

ATQM
Queue manager.

ATUNIX
UNIX application.

ATVMS
Digital OpenVMS 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 WebSphere 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 i5/OS, it is set to AT400; the queue manager never uses ATCICS on i5/OS.

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.

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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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.

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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 below)

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.

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 options listed below 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 back level queue managers. To achieve this, the report option is placed in the MQRO_ACCEPT_UNSUP_MASK subfield.

If a process (either a queue manager or a user process) performs an Activity on a message with MQRO_ACTIVITY 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.

  Refer to the WebSphere MQ Intercommunication book 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 options listed below 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 prior to 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 options listed below 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 option below 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 MQRO_DISCARD_MSG 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 options listed below 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 options listed below 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 or not 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 or not 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 options listed below 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 options listed below 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.

ROPCI

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 ROPCI 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 options listed below 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 Chapter 8, "Report options and message flags," on page 517 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 or not 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 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 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, 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 above. 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 NPMSGDLV 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
MDSID	MDSIDV
MDVER	MDVER2
MDREP	RONONE
MDMT	MTRPRT
MDEXP	EIULIM
MDFB	As appropriate for the nature of the report (FBCOA, FBCOD, FBEXP, or an RC* value)
MDENC	Copied from the original message descriptor
MDCSI	Copied from the original message descriptor
MDFMT	Copied from the original message descriptor
MDPRI	Copied from the original message descriptor
MDPER	Copied from the original message descriptor
MDMID	As specified by the report options in the original message descriptor
MDCID	As specified by the report options in the original message descriptor
MDBOC	0
MDRQ	Blanks
MDRM	Name of queue manager
MDUID	As set by the PMPASI option
MDACC	As set by the PMPASI option
MDAID	As set by the PMPASI option
MDPAT	ATQM, or as appropriate for the message channel agent
MDPAN	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.
MDPD	Date when report message is sent
MDPT	Time when report message is sent
MDAOD	Blanks
MDGID	Copied from the original message descriptor
MDSEQ	Copied from the original message descriptor
MDOFF	Copied from the original message descriptor
MDMFL	Copied from the original message descriptor
MDOLN	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 519.

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 RONONE.

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. For more information about names, see the WebSphere MQ Application Programming Guide.

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 LNQMN. 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; see below) 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 471).

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 56 on page 208. 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 37 on page 100. 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 "Overview" on page 126; also see the WebSphere MQ Application Programming Guide.

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 call

      the 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 regions

the 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 OS/2, the queue manager uses the string "os2".
- On i5/OS, the queue manager uses the name of the user profile associated with the application job.
- On Compaq NonStop Kernel, the queue manager uses the MQSeries principal that is defined for the Tandem user identifier in the MQSeries principal database.
- On HP OpenVMS and 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 and RPG declaration

Table 45. Initial values of fields in MQMD

Field name	Name of constant	Value of constant
MDSID	MDSIDV	'MDbb'
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
MDPT	None	Blanks
MDAOD	None	Blanks
MDGID	GINONE	Nulls
MDSEQ	None	1
MDOFF	None	0
MDMFL	MFNONE	0

Table 45. Initial values of fields in MQMD (continued)

Field name	Name of constant	Value of constant
MDOLN	OLUNDF	-1
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQMDG)

```
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 toidentity
D  MDAID                241    272    INZ
D* Type of application that put themessage
D  MDPAT                273    276I 0 INZ(0)
D* Name of application that put themessage
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 toorigin
D  MDAOD                321    324    INZ
D* Group identifier
D  MDGID                325    348    INZ(X'00000000000000-
D                                     00000000000000000000-
D                                     000000000000')
D* Sequence number of logical messagewithin group
D  MDSEQ                349    352I 0 INZ(1)
D* Offset of data in physical messagefrom 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

The following table summarizes the fields in the structure.

Table 46. Fields in MQMDE

Field	Description	Topic
MESID	Structure identifier	MESID
MEVER	Structure version number	MEVER
MELEN	Length of MQMDE structure	MELEN
MEENC	Numeric encoding of data that follows MQMDE	MEENC
MECSI	Character set identifier of data that follows MQMDE	MECSI
MEFMT	Format name of data that follows MQMDE	MEFMT
MEFLG	General flags	MEFLG
MEGID	Group identifier	MEGID
MESEQ	Sequence number of logical message within group	MESEQ
MEOFF	Offset of data in physical message from start of logical message	MEOFF
MEMFL	Message flags	MEMFL
MEOLN	Length of original message	MEOLN

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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT for the C programming language, respectively.

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

These are described below.

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 48 on page 183.

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 47.

Table 47. Queue-manager action when MQMDE specified on MQPUT or MQPUT1

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 usually 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 47 on page 179 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 the WebSphere MQ Intercommunication manual 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 usually be 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

The MQMDE structure contains the following fields; the fields are described in **alphabetic 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)

Numeric encoding of data that follows MQMDE.

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 125 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 125 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 125. 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 125. 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 125. 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 125. 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 125. 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 and RPG declaration

Table 48. Initial values of fields in MQMDE

Field name	Name of constant	Value of constant
MESID	MESIDV	'MDEb'
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 'b' represents a single blank character.		

RPG declaration (copy file CMQMDEG)

```
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'00000000000000-
D                                     0000000000000000000000-
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

The following table summarizes the fields in the structure.

Table 49. Fields in MQMHBO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options controlling the action of MQMHBUF	Options

Overview for MQMHBO

Availability: All WebSphere MQ systems and WebSphere MQ 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 **alphabetic 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 value. If required values 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).

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 isMBSIDV.

MBVER (10-digit signed integer)

Message handle to buffer options structure - MBVER field

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 and RPG declaration

Message handle to buffer structure - Initial values

Table 50. 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 (copy file MQMHBOG)

```
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 following table summarizes the fields in the structure.

Field	Description	Topic
ODSID	Structure identifier	ODSID

Field	Description	Topic
ODVER	Structure version number	ODVER
ODOT	Object type	ODOT
ODON	Object name	ODON
ODMN	Object queue manager name	ODMN
ODDN	Dynamic queue name	ODDN
ODAU	Alternate user identifier	ODAU
Note: The remaining fields are ignored if *ODVER* is less than ODVER2.		
ODREC	Number of object records present	ODREC
ODKDC	Number of local queues opened successfully	ODKDC
ODUDC	Number of remote queues opened successfully	ODUDC
ODIDC	Number of queues that failed to open	ODIDC
ODORO	Offset of first object record from start of MQOD	ODORO
ODRRO	Offset of first response record from start of MQOD	ODRRO
ODORP	Address of first object record	ODORP
ODRRP	Address of first response record	ODRRP
Note: The remaining fields are ignored if *ODVER* is less than ODVER3.		
ODASI	Alternate security identifier	ODASI
ODRQN	Resolved queue name	ODRQN
ODRMN	Resolved queue manager name	ODRMN
Note: The remaining fields are ignored if *ODVER* is less than ODVER4.		
ODOS	Long object name	ODOS
ODRO	Resolved long object name	ODRO
ODSS	Selection name	ODSS

Overview

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 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. However, if the application is running as an MQ client, the structure must be in the character set and encoding of the client.

Fields

The MQOD structure contains the following fields; the fields are described in **alphabetic 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 or PMALTU 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 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* (see above), except that an asterisk is also valid (see below). A name that is completely blank (or one in which only blanks appear 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 appear 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 above). 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 OOBNDO is specified, the queue manager selects a particular 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 *ODON* 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 manger does *not* belong), *ODMN* should be the name of the queue-sharing group. The name of a queue manager that belongs to that group is also valid, but this is not recommended as it may cause the message to be delayed 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, *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 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 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 i5/OS, 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 TARGTYPE(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.

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.

Either *ODORP* or *ODORO* can be used to specify the object records, but not both; see the description of the *ODORO* field above for details. If *ODORP* 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.

ODOS (MQCHARV)

This specifies the long object name to be used. This field is only referenced 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, as per the description of how to use the MQCHARV structure, then 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.

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 263.

If the object identified by the *ODON* field cannot be found, the call will fail with reason code RC2085 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)

This is the long object name after the queue manager resolves the name provided in *ODON*. This field is only returned 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*, then the value returned in this field is the same as provided in *ODOS*.

If this field is omitted (that is ODRO.VSBufSize is zero) then the *ODRO* will not be returned, but the length will be returned in ODRO.VSLength. If the length is shorter than the full *ODRO* then it will be truncated and will return as many of the rightmost characters as can fit in the provided length.

If *ODRO* is specified incorrectly, as per the description of how to use the MQCHARV structure then the call will fail with reason code RC2520.

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)" 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* above 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 *ODRRO* can be used to specify the response records, but not both; see the description of the *ODRRO* field above 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)

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 MQOT_Q
- If the queue being opened is not being opened using one of the MQOO_INPUT_* options

If *ODSS* is provided in these cases, the call fails with reason code MQRC_SELECTION_NOT_VALID_FOR_TYPE.

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 and RPG declaration

Table 51. Initial values of fields in MQOD

Field name	Name of constant	Value of constant
ODSID	ODSIDV	'ODbb'
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

Table 51. Initial values of fields in MQOD (continued)

Field name	Name of constant	Value of constant
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
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQODG)

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D*
D* MQOD Structure
D*
D* Structure identifier
D  ODSID                  1      4    INZ('OD  ')
D* Structure version number
D  ODVER                  5      8I 0 INZ(1)
D* Object type
D  ODOT                   9     12I 0 INZ(1)
D* Object name
D  ODON                  13     60    INZ
D* Object queue manager name
D  ODMN                  61    108    INZ
D* Dynamic queue name
D  ODDN                 109    156    INZ('AMQ.*')
D* Alternate user identifier
D  ODAU                 157    168    INZ
D* Number of object recordspresent
D  ODREC                169    172I 0 INZ(0)
D* Number of local queues openedsuccessfully
D  ODKDC                173    176I 0 INZ(0)
D* Number of remote queues openedsuccessfully
D  ODUDC                177    180I 0 INZ(0)
D* Number of queues that failed toopen
D  ODIDC                181    184I 0 INZ(0)
D* Offset of first object recordfrom start of MQOD
D  ODORO                185    188I 0 INZ(0)
D* Offset of first response recordfrom start of MQOD
D  ODRRO                189    192I 0 INZ(0)
D* Address of first object record
D  ODORP                193    208*   INZ(*NULL)
D* Address of first responserecord
D  ODRRP                209    224*   INZ(*NULL)
D* Alternate security identifier
D  ODASI                225    264    INZ(X'0000000000000000-
D                                     00000000000000000000-
D                                     00000000000000000000-
D                                     0000000000000000000')
D* Resolved queue name
D  ODRQN                265    312    INZ
D* Resolved queue manager name
D  ODRMN                313    360    INZ
```

```
D* Reserved
D  ODRE1                361    368    INZ
D* Object Long name
D  ODCHRP               369    384*   INZ(*NULL)
D  ODCHRO               385    388I 0 INZ(0)
D  ODCHRS               389    392I 0 INZ(0)
D  ODCHRL               393    396I 0 INZ(0)
D  ODCHRC               397    400I 0 INZ(-3)
D* Message Selector
D  ODCHRPS              401    416*   INZ(*NULL)
D  ODCHROS              417    420I 0 INZ(0)
D  ODCHRSS              421    424I 0 INZ(0)
D  ODCHRLS              425    428I 0 INZ(0)
D  ODCHRCS              429    432I 0 INZ(-3)
D* Resolved Object String
D  ODCHRPR              433    448*   INZ(*NULL)
D  ODCHROR              449    452I 0 INZ(0)
D  ODCHRSR              453    456I 0 INZ(0)
D  ODCHRLR              457    460I 0 INZ(0)
D  ODCHRCR              461    464I 0 INZ(-3)
D* Resolved Object Type
D  ODCHROT              465    468I 0 INZ(0)
```

MQOR – Object record

The following table summarizes the fields in the structure.

Table 52. Fields in MQOR

Field	Description	Topic
ORON	Object name	ORON
ORMN	Object queue manager name	ORMN

Overview

Purpose: The MQOR structure is used 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. 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 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, provided that the queue was opened successfully.

Fields

The MQOR structure contains the following fields; the fields are described in **alphabetic 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 and RPG declaration

Table 53. Initial values of fields in MQOR

Field name	Name of constant	Value of constant
ORON	None	Blanks
ORMN	None	Blanks

RPG declaration (copy file CMQORG)

```
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 following table summarizes the fields in the structure.

Table 54. Fields in MQPD

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options	Options
Support	Required support for message property	Support
Context	Message context to which property belongs	Context
CopyOptions	Copy options to which property belongs	CopyOptions

Overview for MQPD

Availability: AIX, HP-UX, i5/OS, Solaris, Linux, Windows, z/OS and WebSphere MQ 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 **alphabetic order**:

PDCT (10-digit signed integer)

This describes what message context the property belongs to.

When a queue manager receives a message containing a WebSphere 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:

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 WebSphere MQ Version 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:

PDNOC
The property is not associated with a message context.

An unrecognized value is rejected with a *PDREA* 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 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 WebSphere MQ-defined properties; WebSphere MQ sets the appropriate value.

When a queue manager receives a message containing a WebSphere 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, and if you need more than one, the values 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).

COPFOR
This property iscopied 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 MQCOPY_FORWARD, plus MQCOPY_REPORT, plus MQCOPY_PUBLISH.

If none of the options described above is 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 WebSphere MQ-defined properties; support for all other properties is optional.

The field is automatically set to the correct value when the WebSphere 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 a WebSphere MQ-defined property that the queue manager recognizes as being incorrect. the queue manager corrects the value of the *Support* field.

When setting a WebSphere 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 a WebSphere 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 WebSphere MQ-defined properties.

If a WebSphere MQ Version 7.0 queue manager, that supports message properties, receives a property that contains an unrecognized *Support* 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 MQCMHO_NO_VALIDATION 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 WebSphere MQ-defined.

PDSUPR
Support for the property is required. The message is rejected by a queue manager that does not support the WebSphere MQ-defined property. The MQPUT or MQPUT1 call fails with completion code MQCC_FAILED and reason code MQRC_UNSUPPORTED_PROPERTY.

PDSUPL
The message is rejected by a queue manager that does not support the WebSphere 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 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 and RPG declaration

Table 55. 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 (copy file MQPDG)

```
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 following table summarizes the fields in the structure.

Field	Description	Topic
PMSID	Structure identifier	PMSID
PMVER	Structure version number	PMVER
PMOPT	Options that control the action of MQPUT and MQPUT1	PMOPT

Field	Description	Topic
PMTO	Reserved	PMTO
PMCT	Object handle of input queue	PMCT
PMKDC	Number of messages sent successfully to local queues	PMKDC
PMUDC	Number of messages sent successfully to remote queues	PMUDC
PMIDC	Number of messages that could not be sent	PMIDC
PMRQN	Resolved name of destination queue	PMRQN
PMRMN	Resolved name of destination queue manager	PMRMN
Note: The remaining fields are ignored if *PMVER* is less than PMVER2.		
PMREC	Number of put message records or response records present	PMREC
PMPRF	Flags indicating which MQPMR fields are present	PMPRF
PMPRO	Offset of first put-message record from start of MQPMO	PMPRO
PMRRO	Offset of first response record from start of MQPMO	PMRRO
PMPRP	Address of first put message record	PMPRP
PMRRP	Address of first response record	PMRRP
Note: The remaining fields are ignored if *PMVER* is less than PMVER3.		
PMSL	Subscription Level	PMSL

Overview

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 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. However, if the application is running as an MQ client, the structure must be in the character set and encoding of the client.

Fields

The MQPMO structure contains the following fields; the fields are described in **alphabetic 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 neither PMPASI nor PMPASA is 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, 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 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 together (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.

MQPMO_SUPPRESS_REPLYTO

Any information filled into the MDRQ and MDRM fields of the MQMD of this publication will not be 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, this will be 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 may match (at various nodes in the topic tree) and several publications may be sent to the requesting application. See the description of the MQSUBRQ call for more details.

If this option is used and the publication cannot be retained, the message will not be 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 neither this option nor PMNSYP is 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 neither this option nor PMSYP is 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 prior to 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 may 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 extremely 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 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 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 will be more than n physical messages in the group.

PMLOGO
Messages in groups and segments of logical messages will be put in logical order.

This option tells the queue manager how the application will put 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 will use 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.

The above order is called "logical order".

Because the application has told the queue manager how it will put messages in groups and segments of logical messages, the application does not have to maintain and update the group and segment information on 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 and/or current logical message, and the application can terminate them by sending a message (possibly with no application message data) specifying MFLMIG and/or MFLSEG as appropriate, before reissuing the MQPUT call to put the message that is not in the group or not a segment.

Table 56 on page 208 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; "Either" means "Yes" or "No":

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 prior to the call.

Cur log msg
Indicates whether a current logical message exists prior to 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.

MQPMO_RESOLVE_LOCAL_QUEUE
Specifies that the ResolvedQName in the MQPMO structure should be filled in with the name of the local queue which the message actually gets put to. The ResolvedQMgrName will similarly be filled in with the name of the local queue manager hosting the local queue. See MQOO_RESOLVE_LOCAL_QUEUE 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 56. MQPUT options relating to messages in groups and segments of logical messages

Options you specify				Group and log-msg status prior to 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
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

Table 56. MQPUT options relating to messages in groups and segments of logical messages (continued)

Options you specify				Group and log-msg status prior to call		Values the queue manager uses		
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

Notes:

- 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 application's responsibility 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 desired, 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 an CCWARN completion code. Table 57 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 57. 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, prior to 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, should *not* specify PMLOGO. There are two reasons for this:

- If the messages are retrieved and put in order, specifying PMLOGO will cause a new group identifier to be assigned to the messages, and this may 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 may arrive out of order. By specifying neither PMLOGO, nor the corresponding GMLOGO on the MQGET 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.

Applications that generate report messages for messages in groups or segments of logical messages should 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
MDUID	Determined from the environment if possible; set to blanks otherwise.
MDACC	Determined from the environment if possible; set to ACNONE otherwise.
MDAID	Set to blanks.
MDPAT	Determined from the environment.
MDPAN	Determined from the environment if possible; set to blanks otherwise.
MDPD	Set to date when message is put.
MDPT	Set to time when message is put.
MDAOD	Set to blanks.

For more information on message context, see the WebSphere MQ Application Programming Guide.

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 above for values). For more information on message context, see the WebSphere MQ Application Programming Guide.

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 on message context, see the WebSphere MQ Application Programming Guide.

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 above for values). For more information on message context, see the WebSphere MQ Application Programming Guide.

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 on message context, see the WebSphere MQ Application Programming Guide.

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 only one of these options. If neither MQPMO_ASYNC_RESPONSE nor MQPMO_SYNC_RESPONSE are specified, MQPMO_RESPONSE_AS_Q_DEF or MQPMO_RESPONSE_AS_TOPIC_DEF is assumed.

PMARES

The PMARES (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 fields specified above 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 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 require confirmation that messages have been put to a queue you should check both CompCode & 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 may 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 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

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 will behave as for PMARES, and if it is using PMNSYP it will behave 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 above is 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 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 *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 be 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.

PMPRO (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 185.

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 *PMPRO* field above 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* below).

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 LNQMN. 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 185. 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 above 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 will receive 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 and RPG declaration

Table 58. Initial values of fields in MQPMO

Field name	Name of constant	Value of constant
PMSID	PMSIDV	'PMOb'
PMVER	PMVER1	1
PMOPT	PMNONE	0
PMTO	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
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQPMOG)

```
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 ofMQPUT and MQPUT1
D  PMOPT                  9     12I 0 INZ(0)
D* Reserved
D  PMTO                  13     16I 0 INZ(-1)
D* Object handle of input queue
D  PMCT                  17     20I 0 INZ(0)
D* Number of messages sentsuccessfully to local queues
D  PMKDC                 21     24I 0 INZ(0)
D* Number of messages sentsuccessfully to remote queues
D  PMUDC                 25     28I 0 INZ(0)
D* Number of messages that could notbe sent
D  PMIDC                 29     32I 0 INZ(0)
D* Resolved name of destinationqueue
D  PMRQN                 33     80    INZ
D* Resolved name of destination queuemanager
D  PMRMN                 81    128    INZ
D* Number of put message records orresponse records present
D  PMREC                129    132I 0 INZ(0)
D* Flags indicating which MQPMR fieldsare present
D  PMPRF                133    136I 0 INZ(0)
D* Offset of first put message recordfrom start of MQPMO
D  PMPRO                137    140I 0 INZ(0)
D* Offset of first response recordfrom start of MQPMO
```

```
D  PMRRO                141    144I 0 INZ(0)
D* Address of first put messagerecord
D  PMPRP                145    160*   INZ(*NULL)
D* Address of first response record
D  PMRRP                161    176*   INZ(*NULL)
```

MQPMR – Put-message record

The following table summarizes the fields in the structure.

Table 59. Fields in MQPMR

Field	Description	Topic
PRMID	Message identifier	PRMID
PRCID	Correlation identifier	PRCID
PRGID	Group identifier	PRGID
PRFB	Feedback or reason code	PRFB
PRACC	Accounting token	PRACC

Overview

Purpose: The MQPMR structure is used to specify various message properties for a single destination when a message is being put 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. 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, 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

The MQPMR structure contains the following fields; the fields are described in **alphabetic order**:

PRACC (32-byte bit string)

Accounting token.

This is the accounting token to be used for the message sent to the queue whose name 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 125 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 whose name 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 whose name 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 whose name 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 56 on page 208, 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 whose name 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 and RPG declaration

There are no initial values defined for this structure, as no structure declaration is provided. The sample declaration below 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

Overview

Purpose: The MQRFH structure defines the layout of the rules and formatting header. 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

The MQRFH structure contains the following fields; the fields are described in **alphabetic 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.

RFENC (10-digit signed integer)

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, it is recommended that *RFLEN* should be 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 double-quote character; all characters between the open double-quote and the matching close double-quote 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* – see below). However, to assist interoperability an application may prefer to 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-quote characters, the name or value must be enclosed in double quotes, and each double quote 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 and RPG declaration

Table 60. Initial values of fields in MQRFH

Field name	Name of constant	Value of constant
RFSID	RFSIDV	'RFHb'
RFVER	RFVER1	1
RFLEN	RFLENV	32
RFENC	ENNAT	Depends on environment
RFCSI	CSUNDF	0
RFFMT	FMNONE	Blanks
RFFLG	RFNONE	0

Notes:

1. The symbol 'b' represents a single blank character.

RPG declaration (copy file CMQRFHG)

```
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

Overview

Purpose: The MQRFH2 structure defines the format of the version-2 rules and formatting header. 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

The MQRFH2 structure contains the following fields; the fields are described in **alphabetic 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, it is recommended that *RF2LEN* should be 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 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 *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 above, 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 given 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 whose name is the name of 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 below for more information about names.

- `datatype` is an optional user-specified data type of the property; see below for valid data types.
- `value` is the user-specified value of the property; see below 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 whose name is the name of the group:

```
<folder> <group> property1 property2 ... </group> </folder>
```

Groups can be nested within other groups, without limit, and a given 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 name space. 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 name space 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 (see below).
`boolean`	The character 0 or 1 (1 denotes `TRUE`).
`bin.hex`	Hexadecimal digits representing octets.

Data type	Used for
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 wish 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 below. 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
&	&amp;	Mandatory
<	&lt;	Mandatory
>	&gt;	Optional
"	&quot;	Optional
'	&apos;	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 and RPG declaration

Table 61. Initial values of fields in MQRFH2

Field name	Name of constant	Value of constant
RF2SID	RFSIDV	'RFHb'
RF2VER	RFVER2	2
RF2LEN	RFLEN2	36
RF2ENC	ENNAT	Depends on environment
RF2CSI	CSINHT	-2
RF2FMT	FMNONE	Blanks
RF2FLG	RFNONE	0
RF2NVC	None	1208
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQRFH2G)

```
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 followslast NameValueData field
D  RF2ENC                13     16I 0 INZ(273)
```

```
D* Character set identifier of data thatfollows last NameValueData field
D  RF2CSI               17     20I 0 INZ(-2)
D* Format name of data that follows lastNameValueData field
D  RF2FMT               21     28    INZ('        ')
D* Flags
D  RF2FLG               29     32I 0 INZ(0)
D* Character set identifier ofNameValueData
D  RF2NVC               33     36I 0 INZ(1208)
```

MQRMH – Reference message header

The following table summarizes the fields in the structure.

Table 62. Fields in MQRMH

Field	Description	Topic
RMSID	Structure identifier	RMSID
RMVER	Structure version number	RMVER
RMLEN	Total length of MQRMH, including strings at end of fixed fields, but not the bulk data	RMLEN
RMENC	Numeric encoding of bulk data	RMENC
RMCSI	Character set identifier of bulk data	RMCSI
RMFMT	Format name of bulk data	RMFMT
RMFLG	Reference message flags	RMFLG
RMOT	Object type	RMOT
RMOII	Object instance identifier	RMOII
RMSEL	Length of source environment data	RMSEL
RMSEO	Offset of source environment data	RMSEO
RMSNL	Length of source object name	RMSNL
RMSNO	Offset of source object name	RMSNO
RMDEL	Length of destination environment data	RMDEL
RMDEO	Offset of destination environment data	RMDEO
RMDNL	Length of destination object name	RMDNL
RMDNO	Offset of destination object name	RMDNO
RMDL	Length of bulk data	RMDL
RMDO	Low offset of bulk data	RMDO
RMDO2	High offset of bulk data	RMDO2

Overview

Purpose: The MQRMH structure defines the format of a reference message header. This header is used in conjunction 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: 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: 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 *RMLEN* 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 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 will not be converted by the GMCONV option.

Fields

The MQRMH structure contains the following fields; the fields are described in **alphabetic 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, on OS/2 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 *RMLEN* 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 actually 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 actually 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*, 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.

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. It is recommended that the name conform to the same rules as the *RMFMT* field described above.

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, on OS/2 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 *RMLEN* 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 *RMLEN* 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 and RPG declaration

Table 63. Initial values of fields in MQRMH

Field name	Name of constant	Value of constant
RMSID	RMSIDV	'RMHb'
RMVER	RMVER1	1
RMLEN	None	0
RMENC	ENNAT	Depends on environment
RMCSI	CSUNDF	0
RMFMT	FMNONE	Blanks
RMFLG	RMNLST	0
RMOT	None	Blanks
RMOII	OIINON	Nulls
RMSEL	None	0
RMSEO	None	0
RMSNL	None	0
RMSNO	None	0
RMDEL	None	0
RMDEO	None	0
RMDNL	None	0
RMDNO	None	0
RMDL	None	0
RMDO	None	0
RMDO2	None	0
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQRMHG)

```
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, includingstrings 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)
```

MQRR – Response record

The following table summarizes the fields in the structure.

Table 64. Fields in MQRR

Field	Description	Topic
RRCC	Completion code for queue	RRCC
RRREA	Reason code for queue	RRREA

Overview

Purpose: 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. 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively. 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 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

The MQRR structure contains the following fields; the fields are described in **alphabetic 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 whose name 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 whose name 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 and RPG declaration

Table 65. Initial values of fields in MQRR

Field name	Name of constant	Value of constant
RRCC	CCOK	0
RRREA	RCNONE	0

RPG declaration (copy file CMQRRG)

```
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 following table summarizes the fields in the structure.

Table 66. Fields in MQSCO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
KeyRepository	Location of key repository	KeyRepository
CryptoHardware	Details of cryptographic hardware	CryptoHardware
AuthInfoRecCount	Number of MQAIR records present	AuthInfoRecCount

Table 66. Fields in MQSCO (continued)

Field	Description	Topic
AuthInfoRecOffset	Offset of first MQAIR record from start of MQSCO	AuthInfoRecOffset
AuthInfoRecPtr	Address of first MQAIR record	AuthInfoRecPtr
Note: The remaining fields are ignored if *Version* is less than MQSCO_VERSION_2.		
KeyResetCount	SSL secret key reset count	KeyResetCount
Fips Required	Use FIPS-certified cryptographic algorithms in WebSphere MQ	FipsRequired

Overview for MQSCO

Availability: AIX, HP-UX, Solaris, Linux and Windows clients.

Purpose: The MQSCO structure (in conjunction with the SSL fields in the MQCD structure) allows an application running as a WebSphere MQ 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 MQCONNX 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 and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively.

Fields for MQSCO

The MQSCO structure contains the following fields; the fields are described in **alphabetic order**:

SCAIC (10-digit signed integer)

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 11. 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.

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.

  Using *SCAIP* is recommended 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).

  Using *SCAIO* is recommended 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 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 datatype, this field is declared as a byte string of the appropriate length.

SCCH (10-digit signed integer)

This field gives configuration details for cryptographic hardware connected to the client system. Set the field to one of the following strings, or leave it blank or null:

```
GSK_ACCELERATOR_RAINBOW_CS_OFF
GSK_ACCELERATOR_RAINBOW_CS_ON
GSK_ACCELERATOR_NCIPHER_NF_OFF
GSK_ACCELERATOR_NCIPHER_NF_ON
GSK_PKCS11=<the PKCS #11 driver path and filename>;<the PKCS #11
token label>;<the PKCS #11 token password>;<symmetric cipher setting>;
```

Note:

1. The strings containing `RAINBOW` enable or disable the Rainbow Cryptoswift cryptographic hardware.
2. The strings containing `NCIPHER` enable or disable the nCipher nFast cryptographic hardware.
3. In order to use cryptographic hardware which conforms to the PKCS11 interface, for example, the IBM® 4960 or IBM 4963, the PKCS11 driver path, PKCS11 token label, and PKCS11 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 filename is the name of the shared library. An example of the value required for the PKCS #11 path and filename 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, re-configure it with this lowercase label.

4. If the field is blank or null, it indicates that no cryptographic hardware configuration is required.

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_CRYPTO_HARDWARE_ERROR.

This is an input field. The length of this field is given by MQ_SSL_CRYPTO_HARDWARE_LENGTH. The initial value of this field is the null string in C, and blank characters in other programming languages.

SCKR (10-digit signed integer)

This field is relevant only for WebSphere MQ 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 MQRC_KEY_REPOSITORY_ERROR.

To run an SSL connection from a WebSphere MQ 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 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 and RPG declaration

Table 67. Initial values of fields in MQSCO

Field name	Name of constant	Value of constant
SCSID	SCSIDV	'SCOb'
SCVER	SCVER2	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
Notes: 1. The symbol b represents a single blank character.		

RPG declaration (copy file MQSCOG)

```
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)
```

MQSD - Subscription Descriptor

The following table summarizes the fields in the structure.

Field	Description	Topic
SDSID	Structure identifier	SDSID
SDVER	Structure version number	SDVER
SDOPT	Options	SDOPT
SDON	Object name	SDON
SDAU	Alternate User Id	SDAU
SDASI	Alternate Security Id	SDASI
SDOS	Object Long Name	SDOS
SDSN	Subscription Name	SDSN
SDSUD	Subscription user data	SDSUD
SDCID	Subscription Correlation Id	SDCID
SDPRI	Publication priority	SDPRI
SDACC	Publication Accounting Token	SDACC
SDAID	Publication application identity data	SDAID
SDSL	Subscription Level	SDSL

Overview

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.

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 make use of 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 will be sent, by providing an object handle as an output from the MQSUB call. For more information, see "HOBJ (10-digit signed integer) – input/output" on page 428.

The queue manager also undertakes to clean up un-retrieved messages from the managed destination when the subscription is removed, 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 should 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.

Character set and encoding: Data in MQSD must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue-manager attribute and ENNAT, respectively. However, if the application is running as an MQ client, the structure must be in the character set and encoding of the client.

Fields

SDAID (32-byte character string)

This is the value that will be 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 the Message context.

For more information about *MDAID* see MDAID

If the option SOSETI is not specified, the *MDAID* which will be set in each message published for this subscription is blanks, as default context information.

If the option SOSETI 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 MQSO_RESUME, this field is set to the current *MDAID* being used for the subscription.

SDACC (32-byte character string)

This is the value that will be 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 (32-byte bit string)" on page 128

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 option SOSETI 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 will be set in each message published for this subscription.

If the option SOSETI 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 MQSO_RESUME, 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 (40-byte bit string)" on page 187 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 will 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 option SOSCID 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 will be set in each message published for this subscription.

If the option SOSCID 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 will be set in each message published for this subscription will be the correlation identifier created by the original put of the message.

If the option SOGRP 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 SOGRP option.

On return from an MQSUB call using SORES, this field is set to the current correlation id for the subscription.

SDEXP

This is the period of 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 will be 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 263.

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 together. 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 doesn't exist, fails if the subscription already exists.
SORES	Resumes an existing subscription, fails if no subscription exists.
SOCRT + SORES	Creates a subscription if one doesn't exist and resumes a matching one, if it does exist. Useful combination if used in an application that may be run a number of times.
SORES + SOALT (see note)	Resumes an existing subscription, altering any fields to match that specified in the MQSD, fails if no subscription exists.
SOCRT + SOALT (see note)	Creates a subscription if one doesn't exist and resumes a matching one, if it does exist, altering any fields to match that specified in the MQSD. Useful combination if used in an application that wants to ensure its subscription is in a certain state before proceeding.

Combination of options	Notes
Note: Options specifying SOALT can also specify SORES, but this combination has no additional effect to specifying SOALT alone. In other words, SOALT implies SORES, 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.	

SOCRT

Create a new subscription for the topic specified. If a subscription using the same *SDSN* already 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 pre-existing subscription; but if there is no existing subscription, a new one will be created using all the fields provided in the MQSD.

SOCRT can also be combined with SOALT to similar effect (see details about SOALT below).

SORES

Return a handle to a pre-existing subscription which matches that specified by *SDSN*. No changes will be made to the matching subscription's attributes, and they will be 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 above.

The userid of the subscription is the userid that created the subscription, or if it has been subsequently altered by a different userid, it is the userid of the most recent, successful alteration. If an *SDAID* is used, and use of alternate user IDs is allowed for that user, *SDAID* will be recorded as the userid that created the subscription instead of the userid under which the subscription was made.

The userid 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 option SOAUID and the userid 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 will not fail and a handle to the subscription will be returned.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call will fail 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 pre-existing subscription with the full subscription name matching that specified in *SDSN*. Any attributes of the subscription that are different to that specified in the MQSD will be altered in the subscription unless alteration is disallowed for that attribute. Details are noted in the description of each attribute and are summarised in the table below. If you try to alter an attribute that can not be changed, the call fails with the reason code shown in the table below.

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 pre-existing subscription with alterations made as detailed above; but if there is no existing subscription, a new one will be created using all the fields provided in the MQSD.

The userid of the subscription is the userid that created the subscription, or if it has been subsequently altered by a different userid, it is the userid of the most recent, successful alteration. alteration. If *SDAU* is used (and use of alternate user IDs is allowed for that user), then the alternate userid will be recorded as the userid that created the subscription instead of the userid under which the subscription was made.

If a matching subscription exists that was created without the option SOAUID and the userid 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 will not fail and a handle to the subscription will be returned.

If the subscription named in SubName is not a valid subscription to resume or alter from an application, the call will fail 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 on page 255)	RC2515 if you try to alter SOGRP
MQSD	Publication options	Yes (see note 2 on page 255)	None
MQSD	Wildcard options	No	RC2510
MQSD	Other options	No (see note 3 on page 255)	None
MQSD	ObjectName	No	RC2510

Data type descriptor or function call	Field name	Can this attribute be altered using SOALT?	Reason Code
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
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's 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 RC2436.

SONDUR

Request that the subscription to this topic is removed when the application's 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 will 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 will be set.

SOGRP

This subscription is to be 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 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 *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 will cause 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 userid. 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 will cause the call to fail with RC2429.

To add this option to an existing subscription, the MQSUB call, using SOALT, must come from the same userid as the original subscription itself.

If an MQSUB call refers to an existing subscription with SOAUID set, and the userid differs from the original subscription, the call succeeds only if the new userid 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 userid 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 userid to alter the subscription. If the subscription has not been altered, it is the userid that created the subscription.

If an MQSUB verb refers to an existing subscription with SOAUID set and alters the subscription using SOALT to use option SOFUID, the userid of the subscription is now fixed at this new user id. The call succeeds only if the new userid 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 will be 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, will have already been sent to the subscriber as new publications.

If this option is specified without SOCRE it will cause the call to fail with RC2046. On return from an MQSUB call using SORES this option will not be set even if the subscription was created using this option.

If this option is not used, previously retained messages will be 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 will fail.

This option is not valid in combination with SOPUBR.

SOPUBR

Setting this option indicates that the subscriber will request information specifically when required. The queue manager will not to send unsolicited messages to the subscriber. The retained publication (or possibly multiple publications if a wildcard is specified in the topic) will be sent to the subscriber each time a MQSUBRQ call is made using the Hsub handle from a previous MQSUB call. No publications will be sent as a result of the MQSUB call using this option. On return from an MQSUB call using SORES this option will be 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 will be set.

SOWCHR

Wildcards only operate on characters within the topic string.

The behavior defined by SOWCHR is shown in the table below.

Special Character	Behaviour
/	No significance, just another character
*	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?el2/level3/level4
```

Note: This use of wildcards supplies exactly the meaning provided in WebSphere 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	Behaviour
/	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 Brokers 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 will be 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 will be automatically created by the queue manager at subscription time and will be returned to the application in the *SDCID* field. See "SDCID (24-byte bit string)" on page 250 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 OOSETI, 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 will have default context information associated with them as follows:

Field in MQMD	Value used
MDUID	The user id associated with the subscription at the time the subscription was made.
MDACC	Determined from the environment if possible; Set to MQACT_NONE if not.
MDAID	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 will be 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 will be generated for the new user id now owning the subscription and any subsequent publications will be 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.

SDOS (MQCHARV)

This is the long object name to be used.

The *SDOS* 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 263.

The maximum length of *SDOS* is 10240.

If *SDOS* is specified incorrectly, as per the description of how to use the MQCHARV structure or the maximum length is exceeded, then the call will fail 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.

If there are wildcards in the *SDOS* the interpretation of those wildcards can be controlled using the Wildcard options specified in the *SDOPT* 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 *ODRO* field if a buffer is provided.

If altering an existing subscription using the SOALT option, the long name of the topic object subscribed to cannot be changed. This field, and SDON, can be omitted. If they are provided they must resolve to the same full topic name or the call fails with RC2510.

SDPRI (10-digit signed integer)

This is the value that will be 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 (10-digit signed integer)" on page 159.

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 (10-digit signed integer)" on page 159.

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.

ODRO (MQCHARV)

This is the long object name after the queue manager resolves the name provided in *ODON*.

If the long object name is provided in *ODOS* and nothing is provided in *ODON*, then the value returned in this field is the same as provided in *ODOS*.

If this field is omitted (that is ODRO.VSBufSize is zero) then the *ODRO* will not be returned, but the length will be returned in ODRO.VSLength. If the length is shorter than the full *ODRO* then it will be truncated and will return as many of the rightmost characters as can fit in the provided length.

If *ODRO* is specified incorrectly, as per the description of how to use the MQCHARV structure then the call will fail 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

PMPL (10-digit signed integer)

This is the level associated with the subscription. Publications will only be 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)

This specifies the subscription name. This field is only required if *SDOPT* specifies the option SODUR, but if provided will be 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. This is done using the MQSUB call with the SORES option. It is also displayed in the administration view of subscriptions in the *SDSN* field in DISPLAY SBSTATUS.

If *SDSN* is specified incorrectly, as per the description of how to use the MQCHARV structure, or is omitted when it is required (that is *SDSN.VCHRL* is zero), or 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.

SDSUD (MQCHARV)

This specifies the subscription user data. The data provided on the subscription in this field will be 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

Using topic strings

The full topic name is given by the concatenation of two parts. A part exists if the first character of the field is neither a blank nor a null character:

1. The value of the TOPICSTR parameter of the topic object named in *SDON*
2. *SDOS*, if the *VSLength* provided for that variable length string is non-zero

If one of these parts exist it is used unchanged as the topic name.

If both parts they are concatenated in the order they are listed above. A '/' character is inserted between them in the resultant combined topic if one is required.

If neither part exists the call fails with reason code RC2085.

If the full topic name is not valid the call fails with reason code RC2425.

The following table shows examples of topic string concatenation:

TOPICSTR	ObjectString	Concatenation result	Comment
/Football	Scores	/Football/Scores	'/' is added at the concatenation point
/Football/	Scores	/Football/Scores	
/Football	/Scores	/Football/Scores	
/Football/	/Scores	/Football/Scores	'/' is removed at the concatenation point

1. The '/' character is considered to be a special character providing structure to the full topic name. You are recommended not to use the '/' character for any other reason as the structure of the topic tree will not be as you expect. This means that the topic '/Football' is not the same as the topic 'Football'. However, topic '/Football' is the same as the topic '/Football/'.

2. A full topic name with two repeated '/' characters is not valid.
3. If the full topic name is not valid, the call fails with reason code MQRC_TOPIC_STRING_ERROR.
4. Wildcard characters, +, #, * and ? are special characters. You are recommended not to use these characters in your topic strings when publishing. They are not considered invalid however, you should take care to understand the behaviour when using them.
   - Publishing on a topic string with # or + mixed in with other characters (including themselves) within a topic level can be subscribed on, with either wildcard scheme.
   - Publishing on a topic string with # or + as the only character between two '/' characters will produce a topic string that cannot be subscribed on explicitly by an application using the wildcard scheme MQSO_WILDCARD_TOPIC. This will result in the application getting more publications than expected.
   - Publishing on a topic string containing either * or ? anywhere will produce a topic string that cannot be subscribed on explicitly by an application using the wildcard scheme MQSO_WILDCARD_CHAR. This will result in the application getting more publications than expected.

Initial values and RPG declaration

Field name	Name of constant	Value of constant
SDSID	MQSD_STRUC_ID	'SDbb'
SDVER	MQSD_VERSION_1	1
SDOPT	MQSO_NON_DURABLE	0
SDON	None	Null string or blanks
SDAU	None	Null string or blanks
SDASI	MQSID_NONE	Nulls
SDEXP	MQEI_UNLIMITED	-1
SDOS	Names and values as defined for MQCHARV	
SDSN	Names and values as defined for MQCHARV	
SDSUD	Names and values as defined for MQCHARV	
SDCID	MQCI_NONE	Nulls
SDPRI	MQPRI_PRIORITY_AS_Q_DEF	-3
SDACC	MQACT_NONE	Nulls
SDAID	None	Null string or blanks
SDSL	None	1
SDRO	Names and values as defuned in MQCHARV	

Field name	Name of constant	Value of constant
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 MQSD_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure: `MQSD MySD = {MQSD_DEFAULT};`		

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 SDROP                 341    356*
D SDROO                 357    360I 0
D SDROS                 361    364I 0
D SDROL                 365    368I 0
D SDROC                 369    372I 0
```

MQSMPO – Set message property options

The following table summarizes the fields in the structure.

Table 68. Fields in MQSMPO

Field	Description	Topic
StrucId	Structure identifier	StrucId
Version	Structure version number	Version
Options	Options	Options
ValueEncoding	Property value encoding	ValueEncoding
ValueCCSID	Property value character set	ValueCCSID

Overview for MQSMPO

Availability: All WebSphere MQ systems and WebSphere 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 **alphabetic 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 MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQGET or MQPUT call respectively.

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 code MQRC_PROPERTY_NOT_AVAILABLE.

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 MQIMPO_INQ_FIRST or the MQIMPO_INQ_NEXT option.

The property cursor is reset when the message handle is reused, or when the message handle is specified in the *MsgHandle* field of the MQGMO or MQPMO structure on an MQGET or MQPUT call respectively.

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 code MQRC_PROPERTY_NOT_AVAILABLE.

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 MQCCSI_APPL.

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 MQENC_NATIVE.

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 and RPG declaration

Table 69. Initial values of fields in MQSMPO

Field name	Name of constant	Value of constant
SPSID	SPSIDV	'SMPO'
SPVER	SPVER1	1
SPOPT	SPNONE	0
SPVALENC	MQENC_NATIVE	Depends on environment
SPVALCSI	MQCCSI_APPL	-3
Notes: 1. The value Null string or blanks denotes a blank character.		

RPG declaration (copy file MQSMPOG)

```
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*
** 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

Field	Description	Topic
SRSID	Structure identifier	SRSID
SRVER	Structure version number	SRVER
SROPT	Options	SROPT
SRNMP	Number of publications	SRNMP

Overview

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 SRVER1.

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 above 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.

SRVER (10-digit signed integer)

This is the structure version number; the value must be:

SRVER1
Version-1 Subscription Request Options structure.

The following constant specifies the version number of the current version:

SRVERC
Current version of Subscription Request Options structure.

This is always an input field. The initial value of this field is SRVER1.

Initial Values and RPG declaration

Field name	Name of constant	Value of constant
SRSID	SRSIDV	'SROb'
SRVER	SRVER1	1
SROPT	SRNONE	0
SRNMP	None	0

Notes:

1. The symbol b 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 invocation

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQSRO Structure
D*
D* Structure identifier
D  SRSID                  1      4
D* Structure version number
D  SRVER                  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 following table summarizes the fields in the structure.

Table 70. Fields in MQTM

Field	Description	Topic
STSSID	Structure identifier	STSSID
STSVER	Structure version number	STSVER
STSCC	Completion code of first error	STSCC
STSRC	Reason code of first error	STSRC
STSSC	Number of successful asynchronous calls	STSSC
STSWC	Number of asynchronous calls which had warnings	STSWC
STSFC	Number of failed asynchronous calls	STSFC
STSOT	Type of failing object	STSOT
STSOBJN	Name of failing object	STSOBJN
STSOQMGR	Name of queue manager owning the failing object	STSOQMGR
STSROBJN	Resolved name of destination queue	STSROBJN
STSRQMGR	Resolved name of destination queue manager	STSRQMGR

Overview

Purpose: The MQSTS structure describes the data in the status structure returned by 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 *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 425.

Fields

The MQSTS structure contains the following fields; the fields are described in **alphabetic 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 MQCC_OK.

STSPFC (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.

STSOT (10-digit signed integer)

The type of object being named in *ObjectName*. Possible values are:

MQOT_ALIAS_Q
Alias queue.

MQOT_LOCAL_Q
Local queue.

MQOT_MODEL_Q
Model queue.

MQOT_Q
Queue.

MQOT_REMOTE_Q
Remote queue.

MQOT_TOPIC
Topic.

This is always an output field. The initial value of this field is MQOT_Q.

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 MQRC_NONE.

STSROBJN (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, *STSROBJN* 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 *STSROBJN*. *STSRQMGR* can be the name of the local queue manager.

If *STSROBJN* 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, *STSROBJN* 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 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)

This is an output field. The initial value of this field is 48 blank characters.

STSPSC (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:

MQSTS_STRUC_ID
Identifier for status reporting structure.

The initial value of this field is MQSTS_STRUC_ID.

STSVER (10-digit signed integer)

This is the structure version number. The value must be:

MQSTS_VERSION_1
Version number for status reporting structure.

The following constant specifies the version number of the current version:

MQSTS_CURRENT_VERSION
Current version of status reporting structure.

The initial value of this field is MQSTS_VERSION_1.

STSPWC (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 and language declarations

Table 71. Initial values of fields in MQSTS

Field name	Name of constant	Value of constant
STSSID	MQSTS_STRUC_ID	
STSVER	MQSTS_CURRENT_VERSION	MQSTS_VERSION_1
STSCC	MQCC_OK	0
STSRC	MQRC_NONE	0
STSSC	None	0
STSWC	None	0
STSFC	None	0
STSOT	None	0
STSOBJN	None	Blanks
STSOQMGR	None	Blanks
STSROBJN	None	Blanks
STSRQMGR	None	Blanks

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  STSVER                 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  STSOQMGR              81     128
D* Resolved object name
D  STSROBJN             129     176
D* Resolved object queue manager name
D  STSRQMGR             177     224
```

MQTM – Trigger message

The following table summarizes the fields in the structure.

Table 72. Fields in MQTM

Field	Description	Topic
TMSID	Structure identifier	TMSID
TMVER	Structure version number	TMVER
TMQN	Name of triggered queue	TMQN
TMPN	Name of process object	TMPN
TMTD	Trigger data	TMTD
TMAT	Application type	TMAT
TMAI	Application identifier	TMAI
TMED	Environment data	TMED
TMUD	User data	TMUD

Overview

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 WebSphere MQ Trigger Monitor Interface (TMI), which is one of the WebSphere 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 279.

- On i5/OS, the trigger-monitor application provided with WebSphere MQ passes an MQTMC2 structure to the started application.

For information about triggers, see the WebSphere MQ Application Programming Guide.

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
MDSID	MDSIDV
MDVER	MDVER1
MDREP	RONONE
MDMT	MTDGRM
MDEXP	EIULIM
MDFB	FBNONE
MDENC	ENNAT
MDCSI	Queue manager's *CodedCharSetId* attribute
MDFMT	FMTM
MDPRI	Initiation queue's *DefPriority* attribute
MDPER	PENPER
MDMID	A unique value
MDCID	CINONE
MDBOC	0
MDRQ	Blanks
MDRM	Name of queue manager
MDUID	Blanks
MDACC	ACNONE
MDAID	Blanks
MDPAT	ATQM, or as appropriate for the message channel agent
MDPAN	First 28 bytes of the queue manager name
MDPD	Date when trigger message is sent
MDPT	Time when trigger message is sent
MDAOD	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 it 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 **alphabetic 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 468 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 WebSphere 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 468 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 i5/OS 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 468 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 437 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 437 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 437 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 *TMPN* field; see "Attributes for process definitions" on page 468 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 and RPG declaration

Table 73. Initial values of fields in MQTM

Field name	Name of constant	Value of constant
TMSID	TMSIDV	'TMbb'
TMVER	TMVER1	1
TMQN	None	Blanks
TMPN	None	Blanks
TMTD	None	Blanks
TMAT	None	0
TMAI	None	Blanks

Table 73. Initial values of fields in MQTM (continued)

Field name	Name of constant	Value of constant
TMED	None	Blanks
TMUD	None	Blanks
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQTMG)

```
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)

The following table summarizes the fields in the structure.

Table 74. Fields in MQTMC2

Field	Description	Topic
TC2SID	Structure identifier	TC2SID
TC2VER	Structure version number	TC2VER
TC2QN	Name of triggered queue	TC2QN
TC2PN	Name of process object	TC2PN
TC2TD	Trigger data	TC2TD
TC2AT	Application type	TC2AT
TC2AI	Application identifier	TC2AI
TC2ED	Environment data	TC2ED
TC2UD	User data	TC2UD
TC2QMN	Queue manager name	TC2QMN

Overview

Purpose: When a trigger-monitor application retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor may need to pass some or all of the

information in the trigger message to the application that is started by the trigger monitor. 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 WebSphere MQ Trigger Monitor Interface (TMI), which is one of the WebSphere 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 i5/OS, the trigger monitor application provided with WebSphere MQ passes an MQTMC2 structure to the started application.

Fields

The MQTMC2 structure contains the following fields; the fields are described in **alphabetic 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 *TMPN* 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 and RPG declaration

Table 75. Initial values of fields in MQTMC2

Field name	Name of constant	Value of constant
TC2SID	TCSIDV	'TMCb'
TC2VER	TCVER2	'bbb2'
TC2QN	None	Blanks
TC2PN	None	Blanks
TC2TD	None	Blanks
TC2AT	None	Blanks
TC2AI	None	Blanks
TC2ED	None	Blanks
TC2UD	None	Blanks

Table 75. Initial values of fields in MQTMC2 (continued)

Field name	Name of constant	Value of constant
TC2QMN	None	Blanks
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQTMC2G)

```
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 following table summarizes the fields in the structure.

Table 76. Fields in MQWIH

Field	Description	Topic
WISID	Structure identifier	WISID
WIVER	Structure version number	WIVER
WILEN	Length of MQWIH structure	WILEN
WIENC	Numeric encoding of data that follows MQWIH	WIENC
WICSI	Character-set identifier of data that follows MQWIH	WICSI
WIFMT	Format name of data that follows MQWIH	WIFMT
WIFLG	Flags	WIFLG
WISNM	Service name	WISNM
WISST	Service step name	WISST
WITOK	Message token	WITOK
WIRSV	Reserved	WIRSV

Overview

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: 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

The MQWIH structure contains the following fields; the fields are described in **alphabetic 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 and RPG declaration

Table 77. Initial values of fields in MQWIH

Field name	Name of constant	Value of constant
WISID	WISIDV	'WIHb'
WIVER	WIVER1	1
WILEN	WILEN1	120
WIENC	None	0
WICSI	CSUNDF	0
WIFMT	FMNONE	Blanks
WIFLG	WINONE	0

Table 77. Initial values of fields in MQWIH (continued)

Field name	Name of constant	Value of constant
WISNM	None	Blanks
WISST	None	Blanks
WITOK	MTKNON	Nulls
WIRSV	None	Blanks
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQWIHG)

```
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'00000000000000-
D                                     0000000000000000')
D* Reserved
D  WIRSV                 89    120    INZ
```

MQXQH – Transmission-queue header

The following table summarizes the fields in the structure.

Table 78. Fields in MQXQH

Field	Description	Topic
XQSID	Structure identifier	XQSID
XQVER	Structure version number	XQVER
XQRQ	Name of destination queue	XQRQ
XQRQM	Name of destination queue manager	XQRQM
XQMD	Original message descriptor	XQMD

Overview

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 USTRAN.

Format name: FMXQH.

Character set and encoding: Data in MQXQH must be in the character set and encoding of the local queue manager; these are given by the *CodedCharSetId* queue manager attribute and ENNAT for the C programming language, respectively.

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 (see below for details).

  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 – see below for details).

  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: The fields in the separate message descriptor are set by the queue manager as shown below. 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
MDSID	MDSIDV
MDVER	MDVER2

Field in separate MQMD	Value used
MDREP	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.)
MDMT	Copied from the embedded message descriptor.
MDEXP	Copied from the embedded message descriptor.
MDFB	Copied from the embedded message descriptor.
MDENC	ENNAT
MDCSI	Queue manager's *CodedCharSetId* attribute.
MDFMT	FMXQH
MDPRI	Copied from the embedded message descriptor.
MDPER	Copied from the embedded message descriptor.
MDMID	A new value is generated by the queue manager. This message identifier is different from the *MDMID* that the queue manager may have generated for the embedded message descriptor (see above).
MDCID	The *MDMID* from the embedded message descriptor.
MDBOC	0
MDRQ	Copied from the embedded message descriptor.
MDRM	Copied from the embedded message descriptor.
MDUID	Copied from the embedded message descriptor.
MDACC	Copied from the embedded message descriptor.
MDAID	Copied from the embedded message descriptor.
MDPAT	ATQM
MDPAN	First 28 bytes of the queue manager name.
MDPD	Date when message was put on transmission queue.
MDPT	Time when message was put on transmission queue.
MDAOD	Blanks
MDGID	GINONE
MDSEQ	1
MDOFF	0
MDMFL	MFNONE
MDOLN	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, with the exception of 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 indicate the character set and encoding of the character and integer data in the MQXQH structure, respectively. 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 **alphabetic 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 LNQMN. 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 and RPG declaration

Table 79. Initial values of fields in MQXQH

Field name	Name of constant	Value of constant
XQSID	XQSIDV	'XQHb'
XQVER	XQVER1	1
XQRQ	None	Blanks
XQRQM	None	Blanks
XQMD	Same names and values as MQMD; see Table 45 on page 176	–
Notes: 1. The symbol 'b' represents a single blank character.		

RPG declaration (copy file CMQXQHG):

```
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                                     0000000000000000000000-
D                                     000000000000')
D  XQ1CID               177    200    INZ(X'00000000000000-
D                                     0000000000000000000000-
```

```
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'00000000000000-
D                                     00000000000000000000000-
D                                     00000000000000000000000-
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
```

Chapter 2. Function calls

Call descriptions

This chapter describes the MQI calls:

- MQBACK – Back out changes
- MQBEGIN – Begin unit of work
- MQCLOSE – Close object
- MQCMIT – Commit changes
- MQCONN – Connect to queue manager
- MQCONNX – Connect queue
- MQDISC – Disconnect from queue manager
- MQGET – Get message
- MQINQ – Inquire about object attributes
- MQOPEN – Open object
- MQPUT – Put message
- MQPUT1 – Put one message
- MQSET – Set object attributes

Conventions used in the call descriptions

For each call, this chapter 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.

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 1.

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 whose size 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 i5/OS, this call is not supported for applications running in compatibility mode.

Syntax

```
MQBACK (HCONN, COMCOD, REASON)
```

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 MQCONNX 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.

Usage notes

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 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 i5/OS, 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 342 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 prior to 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 prior to 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 may 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 in the event 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 "MQPMO – Put-message options" on page 202, and the GMLOGO option described in "MQGMO – Get-message options" on page 86.

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 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 335 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.

RPG invocation

```
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
```

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, OS/2, i5/OS, Solaris, Windows.

Syntax

MQBEGIN *(HCONN, BEGOP, CMPCOD, REASON)*

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 MQCONNX call.

BEGOP (MQBO) – input/output

Options that control the action of MQBEGIN.

See "MQBO – Begin options" on page 16 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.

Usage notes

1. The MQBEGIN call can be used to start a unit of work that is coordinated by the queue manager and that may 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 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 342 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 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 i5/OS, 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 i5/OS COMMIT and ROLLBACK operations apply to MQ resources as well as to resources belonging to other participating resource managers.

RPG invocation (ILE)

```
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..
DMQBEGIN          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 for MQBUFMH

MQBUFMH *(Hconn, Hmsg, BufMsgHOpts, MsgDesc, Buffer, BufferLength, DataLength, CompCode, Reason)*

Parameters for MQBUFMH

The MQBUFMH call has the following parameters.

HCONN (10-digit signed integer) - output

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 (10-digit signed integer) - output

This is the message handle for which a buffer is required. The value was returned by a previous MQCRTMH call.

BMHOPT (10-digit signed integer) - output

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 14 for details.

MSGDSC (10-digit signed integer) - 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) - output

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 (10-digit signed integer) - output

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 *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

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_FAILED:

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.

Usage notes for MQBUFMH

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.

Language invocations for MQBUFMH

The MQBUFMH call is supported in the programming languages shown below.

C invocation

```
MQBUFMH (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 MQBUFMH */
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 'MQBUFMH' 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(19) BINARY.
**   Options that control the action of MQBUFMH
 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:

```
dcl Hconn        fixed bin(31); /* Connection handle */
dcl Hmsg         fixed bin(63); /* Message handle */
dcl BufMsgHOpts  like MQBMHO;   /* Options that control the action of
                                   MQBUFMH */
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 message buffer */
dcl DataLength   fixed bin(31); /* Length of the output buffer */
dcl CompCode     fixed bin(31); /* Completion code */
dcl Reason       fixed bin(31); /* Reason code qualifying CompCode */
```

System/390 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

Manage callback function

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 function pointer) that is called by WebSphere MQ when certain events occur.

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 cal back 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, with reason MQRC_CALLBACK_FAILED if a callback function does not end normally.

Syntax for MQCB

Message callback function - syntax

```
MQCB (HCONN, OPERATN, HOBJ, CBDSC, MSGDSC,
      GMO, CMPCOD, REASON)
```

Parameters for MQCB

The MQCB call has the following parameters. Manage callback function - 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 MQCONNX call.

On z/OS for CICS applications, and on i5/OS 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.

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 together (do not add the same constant more than once), or
- Combined using the bitwise OR operation (if the programming language supports bit operations).

Combinations that are not valid are noted; all other combinations are valid.

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 whilst 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 in conjunction 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 whilst 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 whilst suspended, and any events missed while in the suspended state are not provided to the operation when it is resumed.

CBDSC (10-digit signed integer) - 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 for details of the structure.

Callback descriptor is required only for the MQCB_REGISTER 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 (MQCBT_EVENT_HANDLER) and should be specified as MQHO_NONE.

If this *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

MSGDSC (10-digit signed integer) - 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 MQGMO_CONVERT option.

See MQMD for details.

MsgDesc is used only 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.

GMO (10-digit signed integer) - 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 86, 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 invoked for messages and events

MQGMO_WAIT with a zero WaitInterval
The MQRC_NO_MSGS_AVAILABLE code is only passed to the consumer when there are no messages and

- the consumer has just 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.

MQGMO_WAIT and a positive WaitInterval
The user is invoked after the specified wait interval with reason code MQRC_NO_MSGS_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_MSGS_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 options are not required the parameter address passed can be null, this is similar to 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.

CMPCOD (10-digit signed integer) - output
Manage callback function - CMPCOD parameter

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_WARNING
Warning (partial completion).

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output
Manage callback function - REASON parameter

The reason codes listed below are the ones that the queue manager can return for the *REASON* parameter.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

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.

Usage notes for MQCB

MQCB function call - 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 afterwards.

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.

This to 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 have 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.

Ensure that your application performs thread-based initialization and cleanup in the START and STOP callbacks. You can do nonthread-based initialization and cleaup 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(MQOP_START_WAIT). This has the effect of 'donating' the thread to 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 MQRC_CALL_IN_PROGRESS.

There are special cases, however, when the application needs to issue a further MQI call before the previous call has completed. For example, the consumer can be invoked during an MQBC call with MQOP_REGISTER.

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 means you can issue, for example, an MQOPEN call, in the consumer function when called with a CallType type of MQCBCT_REGISTER. Any MQI call, with the exception of MQDISC, is allowed.

Language invocations for MQCB

Manage callback function - Language invocations

The MQCB call is supported in the following programming languages.

C invocation

MQCB function call - C language 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 */
```

MQCLOSE - Close object

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN call.

Syntax

MQCLOSE *(HCONN, HOBJ, OPTS, CMPCOD, REASON)*

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 i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

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 whose *DefinitionType* attribute has the value QDPERM (see the *DefinitionType* attribute described in "Attributes for queues" on page 437). 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 un-retrieved 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 behaviour 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 left.

One (and only one) of the following must be specified:

Dynamic queue closure options:

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 80. 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

Table 80. Valid close options for use with retained or deleted objects (continued)

Type of object or queue	CONONE	CODEL	COPURG
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 un-retrieved 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

Task	Subscription closure option
Remove publications on a handle with SOMAN	COPGSB

Default option:

If you require none of the options describes above, 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 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.

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.

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 were 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 OOBRW option, the browse cursor is destroyed. If the queue is subsequently reopened with the OOBRW option, a new browse cursor is created (see the OOBRW 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 86).

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 86.

     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 such time as 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 above, 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 i5/OS, 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.

RPG invocation

```
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
```

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 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 for MQCRTMH

MQCRTMH *(Hconn, CrtMsgHOpts, Hmsg, CompCode, Reason)*

Parameters for MQCRTMH

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 WebSphere 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; WebSphere 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.

On z/OS for CICS applications, and on i5/OS 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

CRTOPT (10-digit signed integer) - input

The options that control the action of MQCRTMH. See MQCMHO for details.

HMSG (10-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 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 will reset 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 WebSphere MQ call at a time. If the handle is in use when a second WebSphere MQ call attempts to use the same message handle, the second WebSphere 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..

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_FAILED:

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 Chapter 5, "Return codes for i5/OS (ILE RPG)," on page 507 for more details.

Usage notes for MQCRTMH

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.
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 i5/OS, 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 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 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 in the event that 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, and the MQGMO_LOGICAL_ORDER option described in MQGMO – Get-message options.

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 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 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.

Language invocations for MQCRTMH

The MQCRTMH call is supported in the programming languages shown below.

C invocation

```
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 invocation

```
     CALL 'MQCRTMH' USING HCONN, CRTMSGOPTS, 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(19) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying COMPCODE
 01  REASON    PIC S9(9) BINARY.
```

PL/I invocation

```
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 */
```

System/390 assembler invocation

```
        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 callback

The MQCTL call performs controlling actions on the object handles opened for a connection. Control callback function

Syntax for MQCTL

Control callback function - syntax

MQCTL *(Hconn, Operation, ControlOpts, CompCode, Reason)*

Parameters for MQCTL

The MQCTL call has the following parameters. Control callback function - parameters

HCONN (10-digit signed integer) - input

Control callback function - HCONN parameter

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 i5/OS 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.

OPERATN (10-digit signed integer) - input

Control callback function - OPERATN parameter

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 - This performs MQCTL with Operation MQOP_STOP before disconnection the HConn.

MQRC_HCONN_ASYNC_ACTIVE is returned if a WebSphere 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 has the effect of stopping the conversation as soon as possible. It is possible, therefore, for a WebSphere MQ API call being issued on the main thread to receive the return code MQRC_HCONN_ASYNC_ACTIVE for a while, followed by the return code MQRC_CONNECTION_BROKEN 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 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 WebSphere 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; at this point 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 period of time, consider closing the queue and reopening it when consumption should 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.

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 MQOP_RESUME is used to resume an MQOP_START_WAIT then the operation blocks.

PCTLOP (10-digit signed integer) - input
Control callback function - PCTLOP parameter

Options that control the action of MQCTL

See MQCTLO for details of the structure.

CMPCOD (10-digit signed integer) - output
Control callback function -CMPCOD parameter

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_WARNING
Warning (partial completion).

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output
Control callback function - REASON parameter

The reason codes listed below are the ones that the queue manager can return for the *Reason* parameter.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_FAILED:

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 an MQOP_REGISTER call.

Or either CallbackFunction or CallbackName have 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 could not be found in the module.

RC2496
(2496, X'9C0') Module is found but is of the wrong type (32bit/64bit) or is not a valid dll.

RC2495
(2495, X'9BF') Module not found in the search path or not authorised 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.

Usage notes for MQCTL

Control callback function - Usage notes

1. Callback routines must check the responses from all services they invoke, and if the routine detects a condition that can not be resolved, it must issue an MQCB MQOP_DEREGISTER command to prevent repeated calls to the callback routine.

Language invocations for MQCTL

Control call backs function - Language invocations

The MQCTL call is supported in the programming languages shown below.

C invocation

MQCTL function call - C language 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 */
```

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 i5/OS, this call is not supported for applications running in compatibility mode.

Syntax

MQCMIT *(HCONN, COMCOD, REASON)*

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 MQCONNX 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.

Usage notes

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 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 i5/OS, 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 342 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 may 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 in the event 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 "MQPMO – Put-message options" on page 202, and the GMLOGO option described in "MQGMO – Get-message options" on page 86.

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 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.

RPG invocation

```
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 i5/OS, 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 i5/OS applications.

  Other applications (that is, applications not running in compatibility mode) must use the MQCONN or MQCONNX call to connect to the queue manager, and the MQDISC call to disconnect from the queue manager. This is the recommended style of programming.

On Websphere MQ for OS/2, Windows, UNIX, and i5/OS, 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

MQCONN *(QMNAME, HCONN, CMPCOD, REASON)*

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 wishes 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 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 i5/OS, 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 usual 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.

MQ client applications: For MQ 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.

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 usual 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 LNQMN.

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 i5/OS, the scope of the handle is the job issuing the call.

On i5/OS 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.

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 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 MQ calls, the application should use the MQDISC call to disconnect from the queue manager.

5. On i5/OS, applications written for releases prior to 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 run-time 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 MQ calls.

   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 MQ calls in single-threaded applications, and for all MQ calls in multi-threaded applications, is provided through the service programs LIBMQM and LIBMQM_R respectively.

7. On i5/OS, 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.

RPG invocation

```
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
```

MQCONNX - Connect queue manager (extended)

The MQCONNX call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent MQ calls.

The MQCONNX call is similar to the MQCONN call, except that MQCONNX allows options to be specified to control the way that the call works.

- On i5/OS, this call is not supported for applications running in compatibility mode.

On Websphere MQ for OS/2, Windows, UNIX, and i5/OS, 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

MQCONNX *(QMNAME, CNOPT, HCONN, CMPCOD, REASON)*

Parameters

The MQCONNX 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 335 for details.

CNOPT (MQCNO) – input/output

Options that control the action of MQCONNX.

See "MQCNO – Connect options" on page 53 for details.

HCONN (10-digit signed integer) – output

Connection handle.

See the *HCONN* parameter described in "MQCONN - Connect queue manager" on page 335 for details.

CMPCOD (10-digit signed integer) – output

Completion code.

See the *CMPCOD* parameter described in "MQCONN - Connect queue manager" on page 335 for details.

CONNID (10-digit signed integer) – output

Connection identifier.

The unique connection identifier associated with an application that is connected to the queue manager (parameter identifier: MQBACF_CONNECTION_ID).

You must specify this parameter or the GenericConnectionId parameter (but not both).

All connections are assigned a unique ID by the queue manager, regardless of how the connection is established. If the connection is established by an MQCONNX with a version 5 MQCNO, the application is able to determine the ConnectionId from the returned MQCNO.

To specify a generic connection identifier, use the GenericConnectionId parameter rather than this one. The only other valid value that ConnectionId can take is that of a specific connection identifier.

The string length of the byte string must be MQ_CONNECTION_ID_LENGTH.

A zero length byte string, or one which contains only null bytes is the same as asking for information about all connection identifiers. This is the only valid value which GenericConnectionId can take. The maximum length of the byte string is MQ_CONNECTION_ID_LENGTH.

REASON (10-digit signed integer) – output

Reason code qualifying *CMPCOD*.

See the *REASON* parameter described in "MQCONN - Connect queue manager" on page 335 for details of possible reason codes.

The following additional reason codes can be returned by the MQCONNX 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 invocation

```
C*..1....:....2....:....3....:....4....:....5....:....6....:....7..
C                     CALLP     MQCONNX(QMNAME : CNOPT : HCONN :
C                                       CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQCONNX          PR                  EXTPROC('MQCONNX')
D* Name of queue manager
D QMNAME                        48A
D* Options that control the action of MQCONNX
D CNOPT                         32A
D* Connection handle
D HCONN                         10I 0
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
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 i5/OS, applications running in compatibility mode do not need to issue this call. See "MQCONN - Connect queue manager" on page 335 for more information.

Syntax

MQDISC *(HCONN, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS 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.

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 i5/OS, applications running in compatibility mode do not have to issue this call; see the MQCONN call for more details.

RPG invocation

```
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 for MQDLTMH

MQDLTMH *(Hconn, Hmsg, DltMsgHOpts, CompCode, Reason)*

Parameters for MQDLTMH

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 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 (10-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:

MQHM_UNUSABLE_HMSG
Unusable message handle.

The message handle cannot be deleted if another MQ call is in progress that was passed the same message handle.

DLTOPT (10-digit signed integer) - input

See MQDMHO for details.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_FAILED:

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 Chapter 5, "Return codes for i5/OS (ILE RPG)," on page 507 for more details.

Usage notes for MQDLTMH

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 297.
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 i5/OS, 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 342 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 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 in the event that 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 202, and the MQGMO_LOGICAL_ORDER option described in "MQGMO – Get-message options" on page 86.

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 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 335 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.

Language invocations for MQDLTMH

The call is supported in the programming languages shown below.

C invocation

Parameters used for the C invocation of MQDLTMH.

```
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

Parameters used for the COBOL invocation of MQDLTMH.

```
     CALL 'MQDLTMH' USING HCONN, HMSG, DLTMSGOPTS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Message handle
 01  HMSG    PIC S9(19) BINARY.
**   Options that control the action of MQDLTMH
 01  DLTMSGHOPTS.
     COPY CMQDMHOV.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying COMPCODE
 01  REASON    PIC S9(9) BINARY.
```

PL/I invocation

Parameters used for the PL/I invocation of MQDLTMH.

```
call MQDLTMH (Hconn, Hmsg, DltMsgHOpts, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn       fixed bin(31);  /* Connection handle */
dcl Hmsg        fixed bin(63);  /* Message handle */
dcl DltMsgHOpts like MQDMHO;    /* Options that control the action of MQDLTMH */
dcl CompCode    fixed bin(31);  /* Completion code */
dcl Reason      fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler invocation

Parameters used for the System/390 assembler invocation of MQDLTMH.

```
        CALL MQDLTMH,(HCONN,HMSG,DLTMSGHOPTS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN          DS        F  Connection handle
HMSG           DS        D  Message handle
DLTMSGHOPTS    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 for MQDLTMP

MQDLTMP *(Hconn, Hmsg, DltPropOpts, Name, CompCode, Reason)*

Parameters for MQDLTMP

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 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 (10-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 (10-digit signed integer) - Input

See the MQDMPO data type for details.

PRNAME (10-digit signed integer) - input

The name of the property to delete. See the WebSphere MQ Application Programming Guide for further information on 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:

MQCC_OK
Successful completion.

MQCC_WARNING
Warning (partial completion).

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_WARNING:

RC2471
(2471, X'09A7') Property not available.

RC2421
(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CMPCOD* is MQCC_FAILED:

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 detailed information on these codes, see:
- WebSphere MQ for z/OS Messages and Codes for WebSphere MQ for z/OS
- WebSphere MQ Messages for all other WebSphere MQ platforms

Language invocations for MQDLTMP

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(19) BINARY.
**   Options that control the action of MQDLTMP
 01  DLTPROPOPTS.
     COPY CMQDMPOV.
**   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 MQDMPO;   /* 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 */
```

System/390 assembler invocation

Parameters used for the System/390 assembler invocation of MQDLTMP.

```
        CALL MQDLTMP,(HCONN,HMSG,DLTPROPOPTS,NAME,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN        DS         F     Connection handle
HMSG         DS         D     Message handle
DLTPROPOPTS  CMQDMPOA   ,     Options that control the action of MQDLTMP
NAME         CMQCHRVA   ,     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, MSGDSC, GMO, BUFLEN, BUFFER, DATLEN, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

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 380 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 125 for details.

If *BUFLEN* is less than the message length, *MSGDSC* is still filled in by the queue manager, whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in "MQGMO – Get-message options" on page 86).

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 86 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 437.

BUFFER (1-byte bit string×BUFLEN) – output

Area to contain the message data.

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 *BUFLEN* is less than the message length, as much of the message as possible is moved into *BUFFER*; this happens whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in "MQGMO – Get-message options" on page 86 for more information).

The character set and encoding of the data in *BUFFER* are given (respectively) by the *MDCSI* and *MDENC* fields returned in the *MSGDSC* parameter. If these 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 86 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, respectively).

If the call fails, the contents of the buffer may 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 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 filled in by the queue manager, whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in "MQGMO – Get-message options" on page 86 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 somewhat 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 reason codes listed below 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 those documented below 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 big for buffer.

RC2190
(2190, X'88E') Converted string too big 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.

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 an GMBRWF or GMBRWN option is specified on the *GMO* parameter (see the *GMOPT* field described in "MQGMO – Get-message options" on page 86).
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 MQ client, it is possible for the message retrieved to be lost if during the processing of the MQGET call the MQ client terminates abnormally or the client connection is severed. This arises because the surrogate that is running on the queue manager's platform and which issues the MQGET call on the client's behalf 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 queue manager's platform, but in this case the window during which a message can be lost is very small. However, as with MQ 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 should wait a short while and then retry the operation.

5. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved provided that certain conditions are satisfied. See the usage notes in the description of the MQPUT call for details. If the conditions are satisfied, the messages will be presented to the receiving application in the order in which they were sent, provided that:
   - 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 could 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 will be presented to the receiving application in the correct order provided that 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 should test for the feedback code FBQUIT in the *MDFB* field of the *MSGDSC* parameter. If this value is found, the application should end. See the *MDFB* field described in "MQMD – Message descriptor" on page 125 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 on message context, see the WebSphere MQ Application Programming Guide.
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 should 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 format names listed below are formats that are converted automatically by the queue manager; these are called "built-in" formats:

FMADMN
FMCICS
FMCMD1
FMCMD2
FMDLH
FMDH
FMEVNT
FMIMS
FMIMVS
FMMDE
FMPCF
FMRMH
FMRFH
FMRFH2
FMSTR
FMTM
FMXQH

- 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 should not begin with the letters "MQ", as such names may 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 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 *may* have been converted successfully; the application should 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 above will be true for conversions performed by user-written exits *only* if the exit conforms to the processing guidelines.

9. For the built-in formats listed above, the queue manager may 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 may 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, provided that 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 should 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 32K bytes.

RPG invocation

```
C*..1....:....2....:....3....:....4....:....5....:....6....:....7..
C                     CALLP     MQGET(HCONN : HOBJ : MSGDSC : GMO :
C                                     BUFLEN : 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                            100A
D* Length in bytes of the BUFFER area
D BUFLEN                          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

MQINQ *(HCONN, HOBJ, SELCNT, SELS, IACNT, INTATR, CALEN, CHRATR, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

HOBJ (10-digit signed integer) – input

Object handle.

This handle represents the object (of any type) whose attributes 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×SELCNT) – input

Array of attribute selectors.

This is an array of *SELCNT* attribute selectors; each selector identifies an attribute (integer or character) whose value 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 will accept.
2. If all of 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 81. 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 re-queue 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).	

Table 81. MQINQ attribute selectors for queues (continued). See the bottom of the table for an explanation of the notes.

Selector	Description	Note
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.	
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.	

Table 81. MQINQ attribute selectors for queues (continued). See the bottom of the table for an explanation of the notes.

Selector	Description	Note
IAUSAG	Usage.	
CLWLUSEQ	Use remote queues.	

Notes:

1. Supported on AIX, HP-UX, z/OS, OS/2, i5/OS, Solaris, Windows, plus WebSphere MQ clients connected to these systems.
2. Supported on AIX, HP-UX, OS/2, i5/OS, Solaris, Windows, plus WebSphere MQ clients connected to these systems.
3. Supported on z/OS.
4. Not supported on z/OS.
5. Not supported on VSE/ESA.

Table 82. MQINQ attribute selectors for namelists. See the bottom of Table 81 on page 362 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 × Number of names in the list).	1
IANAMC	Number of names in the namelist.	1
IAQSGD	Queue-sharing group disposition.	3

Table 83. MQINQ attribute selectors for process definitions. See the bottom of Table 81 on page 362 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
CAENVD	Environment data (LNPROE).	5
CAPROD	Description of process definition (LNPROD).	5
CAPRON	Name of process definition (LNPRON).	5
CAUSRD	User data (LNPROU).	5
IAAPPT	Application type.	5
IAQSGD	Queue-sharing group disposition.	3

Table 84. MQINQ attribute selectors for the queue manager. See the bottom of Table 81 on page 362 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

Table 84. MQINQ attribute selectors for the queue manager (continued). See the bottom of Table 81 on page 362 for an explanation of the notes.

Selector	Description	Note
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
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
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	Syncpoint availability.	5
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×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×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 big.

RC2071
(2071, X'817') Insufficient storage available.

RC2195
(2195, X'893') Unexpected error occurred.

Usage notes

1. The values returned are a snapshot of the selected attributes. There is no guarantee that the attributes will not change 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 subsequently 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 86 on page 438 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 attributes listed below 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

may inquire different instances of the cluster queue, although usually all of the instances have the same attribute values.

For more information about cluster queues, refer to the WebSphere MQ Queue Manager Clusters book.

5. If a number of attributes are to be inquired, and subsequently some of them are to be set using the MQSET call, it may 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 437
   - "Attributes for namelists" on page 466
   - "Attributes for process definitions" on page 468
   - "Attributes for the queue manager" on page 471
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).

RPG invocation

```
C*..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 for MQINQMP

MQINQMP *(Hconn, Hmsg, InqPropOpts, Name, PropDesc, Type, ValueLength, Value, DataLength, CompCode, Reason)*

Parameters for MQINQMP

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 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 (10-digit signed integer) - input

This is the message handle to be inquired. The value was returned by a previous MQCRTMH call.

INQOPT (10-digit signed integer) - Input

See the *WebSphere MQ Application Programming Reference* data type for details.

PRNAME (10-digit signed integer) - input

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 '%' at the end of the property name. The wildcard matches zero or more characters, including the '.' 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 filled in on return from MQINQMP with the same 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 with the MQ property name e.g. “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 all properties that begin ’usr’, respectively:

MQPROP_INQUIRE_ALL
Inquire on all properties of the message.

MQPRP_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.

PRPDSC (10-digit signed integer) - 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:

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 *i5/OS Application Programming Reference (ILE RPG)* 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.

VALUE (10-digit signed integer) - 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 may 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 the parameter address passed by programs written in C or System/390 assembler is null.

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 MQIMPO_QUERY_LENGTH option has been specified; in this case no value is returned.

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 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 may 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.

CMPCOD (10-digit signed integer) - output

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_WARNING
Warning (partial completion).

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - 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_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 big 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 big 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 on these codes, see:
- WebSphere MQ for z/OS Messages and Codes for WebSphere MQ for z/OS

- WebSphere MQ Messages for all other WebSphere MQ platforms

Language invocations for MQINQMP

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 */
MQDIMPO 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, PROPDESC, 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(19) BINARY.
**   Options that control the action of MQINQMP
 01  INQMSGOPTS.
     COPY CMQIMPOV.
**   Property name
 01  NAME.
     COPY CMQCHRVV.
**   Property descriptor
 01  PROPDESC.
     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:

```
dcl Hconn       fixed bin(31);  /* Connection handle */
dcl Hmsg        fixed bin(63);  /* Message handle */
dcl InqPropOpts like MQIMPO;    /* Options that control the action of MQINQMP */
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 in bytes of the Value area */
dcl Value       char (n);       /* Area to contain the property value */
dcl DataLength  fixed bin (31); /* Length of the property value */
dcl CompCode    fixed bin (31); /* Completion code */
dcl Reason      fixed bin (31); /* Reason code qualifying CompCode */
```

System/390 assembler invocation

Parameters used for the System/390 assembler invocation of MQINQMP.

```
CALL MQINQMP,(HCONN,HMSG,INQMSGOPTS,NAME,PROPDESC,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
PROPDESC     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 converts a message handle into a buffer and is the inverse of the MQBUFMH call.

Syntax for MQMHBUF

MQMHBUF *(Hconn, Hmsg, MsgHBufOpts, Name, MsgDesc, BufferLength, Buffer, DataLength, CompCode, Reason)*

Parameters for MQMHBUF

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 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 (10-digit signed integer) - input

This is the message handle for which a buffer is required.

The value was returned by a previous MQCRTMH call.

MHBOPT (10-digit signed integer) - 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 14 for details.

PRNAME (10-digit signed integer) - 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.

Wildcards

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', respectively:

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.

MSGDSC (10-digit signed integer) - 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

BUFFLEN is the length of the Buffer area, in bytes.

BUFFER (10-digit signed integer) - output

BUFFER defines the area to contain the message properties. You should align the buffer on a 4-byte boundary.

If *BUFFLEN* 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.

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 MQRC_PROPERTY_NOT_AVAILABLE.

If *BUFLEN* 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 *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:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_FAILED:

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.

Usage notes for MQMHBUF

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.

Language invocations for MQMHBUF

The MQMHBUF call is supported in the programming languages shown below.

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 */
```

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(19) BINARY.
**   Options that control the action of MQMHBUF
 01  MSGHBUFOPTS.
     COPY CMQMHBOV.
**   Property name
 01  NAME
     COPY CMQCHRVV.
```

```
**   Message descriptor
 01  MSGDESC
     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 */
```

System/390 assembler invocation

```
CALL MQMHBUF,(HCONN,HMSG,MSGHBUFOPTS,NAME,MSGDESC,BUFFERLENGTH,
              BUFFER,DATALENGTH,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN         DS       F      Connection handle
HMSG          DS       D      Message handle
MSGHBUFOPTS   CMQMHBOA ,      Options that control the action of MQMHBUF
NAME          CMQCHRVA ,      Property name
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 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

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

OBJDSC (MQOD) – input/output

Object descriptor.

This is a structure that identifies the object to be opened; see "MQOD – Object descriptor" on page 185 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 437). 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

See below for details of these options; other options can be specified as required. If more than one option is required, the values can be added together (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" on page 386).

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 437 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 86 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.

OOSET
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 a particular instance of the cluster queue:

OOBNDO
Bind handle to destination when queue is opened.

This causes the local queue manager to bind the queue handle to a particular 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 a particular 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 subsequently 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 OOBRW 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 usually 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 neither OOBNDO nor OOBNDN is specified.

OOBNDQ is defined to aid program documentation. It is not intended that this option be 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 on message context, see the WebSphere MQ Application Programming Guide.

This context information can be passed to a message that is subsequently put on a queue using the MQPUT or MQPUT1 calls. See the PMPASI and PMPASA options described in "MQPMO – Put-message options" on page 202.

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 on message context, see the WebSphere MQ Application Programming Guide.

The OOOUT 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 on message context, see the WebSphere MQ Application Programming Guide.

This option implies OOPASI, which need not therefore be specified. The OOOUT 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 on message context, see the WebSphere MQ Application Programming Guide.

This option implies OOPASI, which need not therefore be specified. The OOOUT 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 on message context, see the WebSphere MQ Application Programming Guide.

This option implies the following options, which need not therefore be specified:

- OOPASI
- OOPASA
- OOSETI

The OOOUT 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.

OORLQ
Fills in the name of local queue that was opened.

This option specifies that the ResolvedQName in the MQOD structure (if available) should be filled in with the name of the local queue which was actually opened. The ResolvedQMgrName will similarly be filled in with the name of the local queue manager hosting the local queue.

Valid MQOPEN options for each queue type

Option	Alias (note 1)	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
OOINPQ	Y	Y	—	—	—	—
OOINPS	Y	Y	—	—	—	—
OOINPX	Y	Y	—	—	—	—
OOBRW	Y	Y	—	—	—	—
OOOUT	Y	Y	Y	Y	Y	Y
OOINQ	Y	Y	Note 2	Y	—	—
OOSET	Y	Y	Note 2	—	—	—
OOBNDO (note 3)	Y	Y	Y	Y	Y	—
OOBNDN (note 3)	Y	Y	Y	Y	Y	—

Option	Alias (note 1)	Local and Model	Remote	Nonlocal Cluster	Distribution list	Topic
OOBNDQ (note 3)	Y	Y	Y	Y	Y	—
OOSAVA	Y	Y	—	—	—	—
OOPASI	Y	Y	Y	Y	Y	Note 5
OOPASA	Y	Y	Y	Y	Y	Note 5
OOSETI	Y	Y	Y	Y	Y	Note 5
OOSETA	Y	Y	Y	Y	Y	Note 5
OOALTU	Y	Y	Y	Y	Y	Y
OOFIQ	Y	Y	Y	Y	Y	Y
OORLQ	Y	Y	Y	Y	—	—

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 i5/OS, 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.

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:

RC2136
(2136, X'858') Multiple reason codes returned.

If *CMPCOD* is CCFAIL:

RC2001
(2001, X'7D1') Alias base queue not a valid type.

RC2219
(2219, X'8AB') MQI call reentered before previous call complete.

RC2266
(2266, X'8DA') Cluster workload exit failed.

RC2268
(2268, X'8DC') Put calls inhibited for all queues in cluster.

RC2189
(2189, X'88D') Cluster name resolution failed.

RC2269
(2269, X'8DD') Cluster resource error.

RC2009
(2009, X'7D9') Connection to queue manager lost.

RC2198
(2198, X'896') Default transmission queue not local.

RC2199
(2199, X'897') Default transmission queue usage error.

RC2011
(2011, X'7DB') Name of dynamic queue not valid.

RC2017
(2017, X'7E1') No more handles available.

RC2018
(2018, X'7E2') Connection handle not valid.

RC2019
(2019, X'7E3') Object handle not valid.

RC2194
(2194, X'892') Object name not valid for object type.

RC2035
(2035, X'7F3') Not authorized for access.

RC2100
(2100, X'834') Object already exists.

RC2101
(2101, X'835') Object damaged.

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.

RC2045
(2045, X'7FD') Option not valid for object type.

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.

RC2057
(2057, X'809') Queue type not valid.

RC2184
(2184, X'888') Remote queue name not valid.

RC2102
(2102, X'836') Insufficient system resources available.

RC2063
(2063, X'80F') Security error occurred.

RC2188
(2188, X'88C') Call rejected by cluster workload exit.

RC2071
(2071, X'817') Insufficient storage available.

RC2195
(2195, X'893') Unexpected error occurred.

RC2082
(2082, X'822') Unknown alias base queue.

RC2197
(2197, X'895') Unknown default transmission queue.

RC2085
(2085, X'825') Unknown object name.

RC2086
(2086, X'826') Unknown object queue manager.

RC2087
(2087, X'827') Unknown remote queue manager.

RC2196
(2196, X'894') Unknown transmission queue.

RC2091
(2091, X'82B') Transmission queue not local.

RC2092
(2092, X'82C') Transmission queue with wrong usage.

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. See the *OBJDSC* parameter described in "MQOPEN - Open object" on page 380.

     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 may include one or more of the following for a given 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:

- OOBNDO
- OOBNDN
- OOBNDQ

Refer to the WebSphere MQ Queue Manager Clusters book 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 this is the case, *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 subsequently 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 185):

   - 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 may reject it because the user originating the message is not authorized.

7. An MQOPEN call with the OOBRW 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 subsequently 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.

   a. 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.

b. Only the following open options are valid in the *OPTS* parameter:
   - OOOUT
   - OOPAS*
   - OOSET*
   - OOALTU
   - OOFIQ

c. 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.

d. 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 and reason code are set to CCOK and RCNONE respectively; 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.

e. 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.

f. 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.

g. 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.

h. 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.

i. It is valid for a distribution list to contain only one destination.

9. The following notes apply to the use of cluster queues.

   a. 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 may 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.

   b. 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:
        - OOBRW
        - OOINPQ
        - OOINPX
        - OOINPS
        - OOSET

        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
        - OOOUT

        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, refer to the WebSphere MQ Queue Manager Clusters book.

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 i5/OS, 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.

12. When using the MQOO_RESOLVE_LOCAL_QUEUE 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 filled in with the RemoteQName and RemoteQMgrName found in the remote queue definition, or similarly with the chosen remote cluster queue. If MQOO_RESOLVE_LOCAL_QUEUE is specified when opening, for example, a remote queue, ResolvedQName will now be the transmission queue which messages will be actually put to. The ResolvedQMgrName will be filled in 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.

RPG invocation

```
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                        456A
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

MQPUT *(HCONN, HOBJ, MSGDSC, PMO, BUFLEN, BUFFER, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

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 125 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 178 for more details.

PMO (MQPMO) – input/output

Options that control the action of MQPUT.

See "MQPMO – Put-message options" on page 202 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 above conditions is 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×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 and/or numeric data, 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 and encoding of the local queue manager (given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively).

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:

RC2136
(2136, X'858') Multiple reason codes returned.

RC2049
(2049, X'801') Message Priority exceeds maximum value supported.

RC2104
(2104, X'838') Report option(s) in message descriptor not recognized.

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.

RC2266
(2266, X'8DA') Cluster workload exit failed.

RC2189
(2189, X'88D') Cluster name resolution failed.

RC2269
(2269, X'8DD') Cluster resource error.

RC2009
(2009, X'7D9') Connection to queue manager lost.

RC2097
(2097, X'831') Queue handle referred to does not save context.

RC2098
(2098, X'832') Context not available for queue handle referred to.

RC2135
(2135, X'857') Distribution header structure not valid.

RC2013
(2013, X'7DD') Expiry time not valid.

RC2014
(2014, X'7DE') Feedback code not valid.

RC2258
(2258, X'8D2') Group identifier 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.

RC2185
(2185, X'889') Inconsistent persistence specification.

RC2245
(2245, X'8C5') Inconsistent unit-of-work specification.

RC2026
(2026, X'7EA') Message descriptor not valid.

RC2248
(2248, X'8C8') Message descriptor extension not valid.

RC2027
(2027, X'7EB') Missing reply-to queue.

RC2249
(2249, X'8C9') Message flags not valid.

RC2250
(2250, X'8CA') Message sequence number 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.

RC2029
(2029, X'7ED') Message type in message descriptor not valid.

RC2136
(2136, X'858') Multiple reason codes returned.

RC2270
(2270, X'8DE') No destination queues available.

RC2479
(2479, X'9AF') Publication could not be retained.

RC2039
(2039, X'7F7') Queue not open for output.

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.

RC2041
(2041, X'7F9') Object definition changed since opened.

RC2101
(2101, X'835') Object damaged.

RC2251
(2251, X'8CB') Message segment offset not valid.

RC2137
(2137, X'859') Object not opened successfully.

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2252
(2252, X'8CC') Original length not valid.

RC2149
(2149, X'865') PCF structures not valid.

RC2047
(2047, X'7FF') Persistence not valid.

RC2048
(2048, X'800') Queue does not support persistent messages.

RC2173
(2173, X'87D') Put-message options structure not valid.

RC2158
(2158, X'86E') Put message record flags not valid.

RC2050
(2050, X'802') Message priority not valid.

RC2502
(2502, X'9C6') Publication failed, and publication has not been delivered to any subscribers

RC2051
(2051, X'803') Put calls inhibited for the queue.

RC2159
(2159, X'86F') Put message records not valid.

RC2052
(2052, X'804') Queue has been deleted.

RC2053
(2053, X'805') Queue already contains maximum number of messages.

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.

RC2056
(2056, X'808') No space available on disk for queue.

RC2154
(2154, X'86A') Number of records present not valid.

RC2061
(2061, X'80D') Report options in message descriptor not valid.

RC2156
(2156, X'86C') Response records not valid.

RC2102
(2102, X'836') Insufficient system resources available.

RC2253
(2253, X'8CD') Length of data in message segment is zero.

RC2188
(2188, X'88C') Call rejected by cluster workload exit.

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.

RC2480
(2480, X'9B0') Target type has changed: the alias queue referred to a queue but now refers to a topic.

RC2195
(2195, X'893') Unexpected error occurred.

RC2255
(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2257
(2257, X'8D1') Wrong version of MQMD supplied.

RC2420
(2420) An MQPUT call was issued, but the message data contains an MQEPH structure that is not valid.

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 behaviour is dependant 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 were 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 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.

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, thereby 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 provided that the

conditions detailed below 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.

  Be aware that when messages are put onto a particular queue within a single unit of work, messages from other applications may 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 i5/OS, 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 and reason code are set to CCOK and RCNONE respectively; 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 437).

   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 big 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 may 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 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 MQ header structures are validated completely by the queue manager: MQDH, MQMDE.

For other 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 MQ structures, the following conditions must be satisfied:

- An 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
- 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.

RPG invocation

```
C*..1....:....2....:....3....:....4....:....5....:....6....:....7..
C                   CALLP     MQPUT(HCONN : HOBJ : MSGDSC : PMO :
C                                   BUFLEN : 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                           176A
D* Length of the message in BUFFER
D BUFLEN                         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 queueor distribution list, or to a topic. The queue, distribution list, or topic does not need to be open.

Syntax

MQPUT1 *(HCONN, OBJDSC, MSGDSC, PMO, BUFLEN, BUFFER, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

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 185 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 125 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 178 for more details.

PMO (MQPMO) – input/output

Options that control the action of MQPUT1.

See "MQPMO – Put-message options" on page 202 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×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 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 and/or numeric data, 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 and encoding of the local queue manager (given by the *CodedCharSetId* queue manager attribute and ENNAT, respectively).

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:

RC2136
(2136, X'858') Multiple reason codes returned.

RC2241
(2241, X'8C1') Message group not complete.

RC2242
(2242, X'8C2') Logical message not complete.

RC2049
(2049, X'801') Message Priority exceeds maximum value supported.

RC2104
(2104, X'838') Report option(s) in message descriptor not recognized.

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.

RC2219
(2219, X'8AB') MQI call reentered before previous call complete.

RC2266
(2266, X'8DA') Cluster workload exit failed.

RC2189
(2189, X'88D') Cluster name resolution failed.

RC2269
(2269, X'8DD') Cluster resource error.

RC2009
(2009, X'7D9') Connection to queue manager lost.

RC2097
(2097, X'831') Queue handle referred to does not save context.

RC2098
(2098, X'832') Context not available for queue handle referred to.

RC2198
(2198, X'896') Default transmission queue not local.

RC2199
(2199, X'897') Default transmission queue usage error.

RC2135
(2135, X'857') Distribution header structure not valid.

RC2013
(2013, X'7DD') Expiry time not valid.

RC2014
(2014, X'7DE') Feedback code not valid.

RC2258
(2258, X'8D2') Group identifier not valid.

RC2017
(2017, X'7E1') No more handles available.

RC2018
(2018, X'7E2') Connection handle not valid.

RC2026
(2026, X'7EA') Message descriptor not valid.

RC2248
(2248, X'8C8') Message descriptor extension not valid.

RC2027
(2027, X'7EB') Missing reply-to queue.

RC2249
(2249, X'8C9') Message flags not valid.

RC2250
(2250, X'8CA') Message sequence number 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.

RC2029
(2029, X'7ED') Message type in message descriptor not valid.

RC2136
(2136, X'858') Multiple reason codes returned.

RC2270
(2270, X'8DE') No destination queues available.

RC2035
(2035, X'7F3') Not authorized for access.

RC2101
(2101, X'835') Object damaged.

RC2042
(2042, X'7FA') Object already open with conflicting options.

RC2155
(2155, X'86B') Object records not valid.

RC2043
(2043, X'7FB') Object type not valid.

RC2044
(2044, X'7FC') Object descriptor structure not valid.

RC2251
(2251, X'8CB') Message segment offset not valid.

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2252
(2252, X'8CC') Original length not valid.

RC2149
(2149, X'865') PCF structures not valid.

RC2047
(2047, X'7FF') Persistence not valid.

RC2048
(2048, X'800') Queue does not support persistent messages.

RC2173
(2173, X'87D') Put-message options structure not valid.

RC2158
(2158, X'86E') Put message record flags not valid.

RC2050
(2050, X'802') Message priority not valid.

RC2051
(2051, X'803') Put calls inhibited for the queue.

RC2159
(2159, X'86F') Put message records not valid.

RC2052
(2052, X'804') Queue has been deleted.

RC2053
(2053, X'805') Queue already contains maximum number of messages.

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.

RC2056
(2056, X'808') No space available on disk for queue.

RC2057
(2057, X'809') Queue type not valid.

RC2154
(2154, X'86A') Number of records present not valid.

RC2184
(2184, X'888') Remote queue name not valid.

RC2061
(2061, X'80D') Report options in message descriptor not valid.

RC2102
(2102, X'836') Insufficient system resources available.

RC2156
(2156, X'86C') Response records not valid.

RC2063
(2063, X'80F') Security error occurred.

RC2253
(2253, X'8CD') Length of data in message segment is zero.

RC2188
(2188, X'88C') Call rejected by cluster workload exit.

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.

RC2082
(2082, X'822') Unknown alias base queue.

RC2197
(2197, X'895') Unknown default transmission queue.

RC2085
(2085, X'825') Unknown object name.

RC2086
(2086, X'826') Unknown object queue manager.

RC2087
(2087, X'827') Unknown remote queue manager.

RC2196
(2196, X'894') Unknown transmission queue.

RC2255
(2255, X'8CF') Unit of work not available for the queue manager to use.

RC2257
(2257, X'8D1') Wrong version of MQMD supplied.

RC2091
(2091, X'82B') Transmission queue not local.

RC2092
(2092, X'82C') Transmission queue with wrong usage.

RC2420
(2420) An MQPUT1 call was issued, but the message data contains an MQEPH structure that is not valid.

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, thereby 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 provided that 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 MQRR 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 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.

RPG invocation

```
C*..1....:....2....:....3....:....4....:....5....:....6....:....7..
C                     CALLP     MQPUT1(HCONN : OBJDSC : MSGDSC :
C                                      PMO : BUFLEN : 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                        360A
D* Message descriptor
D MSGDSC                        364A
D* Options that control the action of MQPUT1
D PMO                           176A
D* Length of the message in BUFFER
D BUFLEN                         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

MQSET *(HCONN, HOBJ, SELCNT, SELS, IACNT, INTATR, CALEN, CHRATR, CMPCOD, REASON)*

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

HOBJ (10-digit signed integer) – input

Object handle.

This handle represents the queue object whose attributes are to be set. The handle was returned by a previous MQOPEN call that specified the OOSET 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×SELCNT) – input

Array of attribute selectors.

This is an array of *SELCNT* attribute selectors; each selector identifies an attribute (integer or character) whose value 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 below.

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 given 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 given in parentheses.

Table 85. MQSET attribute selectors for queues

Selector	Description	Note
CATRGD	Trigger data (LNTRGD).	2
IADIST	Distribution list support.	1
IAIGET	Whether get operations are allowed.	
IAIPUT	Whether put operations are allowed.	
IATRGC	Trigger control.	2
IATRGD	Trigger depth.	2
IATRGP	Threshold message priority for triggers.	2
IATRGT	Trigger type.	2

Notes:

1. Supported only on AIX, HP-UX, OS/2, i5/OS, Solaris, Windows, plus WebSphere 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×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×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 big.

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.

Usage notes

1. Using this call, the application can specify an array of integer attributes, or a collection of character attribute strings, or both. The attributes specified are all set simultaneously, if no errors occur. If an error does occur (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 361 for details.

   Note: Not all attributes whose values 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. It is not possible to change the attributes of a model queue using the MQSET call. However, if you open a model queue using the MQOPEN call with the OOSET 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.
6. Changes to attributes resulting from use of the MQSET call do not affect the values of the *AlterationDate* and *AlterationTime* attributes.
7. For more information about object attributes, see:
   - "Attributes for queues" on page 437
   - "Attributes for namelists" on page 466
   - "Attributes for process definitions" on page 468
   - "Attributes for the queue manager" on page 471

RPG invocation

```
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
D* Reason code qualifying CMPCOD
D REASON                        10I 0
```

MQSETMP – Set message handle property

Call that sets a property of a message handle

The MQSETMP call sets or modifies a property of a message handle.

Syntax for MQSETMP

MQSETMP call syntax and list of parameters

MQSETMP *(Hconn, Hmsg, SetPropOpts, Name, PropDesc, Type, ValueLength, Value, CompCode, Reason)*

Parameters for MQSETMP

List of valid parameters for the MQSETMP call.

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 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 (10-digit signed integer) - input/output

This is the message handle to be modified. The value was returned by a previous MQCRTMH call.

SETOPT (10-digit signed integer) - 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 (10-digit signed integer) - 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 (10-digit signed integer) - 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:

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.

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 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 (10-digit signed integer) - 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:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) - output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* is MQCC_WARNING:

RC2421
(2421, X'0975') An MQRFH2 folder containing properties could not be parsed.

If *CMPCOD* is MQCC_FAILED:

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 Chapter 5, "Return codes for i5/OS (ILE RPG)," on page 507 for more details.

Usage notes for MQSETMP

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 297.
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 i5/OS, 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 342 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 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 in the event that 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 202, and the MQGMO_LOGICAL_ORDER option described in "MQGMO – Get-message options" on page 86.

   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 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 335 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.

Language invocations for MQSETMP

The MQSETMP call is supported in the programming languages shown below. List of languages supporting the MQSETMP call

C invocation

Parameters used for the C invocation of MQSETMP.

```
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

Parameters used for the COBOL invocation of MQSETMP.

```
     CALL 'MQSETMP' USING HCONN, HMSG, SETMSGOPTS, NAME, PROPDESC, TYPE,
                          VALUELENGTH, VALUE, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Message handle
 01  HMSG    PIC S9(19) BINARY.
**   Options that control the action of MQSETMP
 01  SETMSGOPTS.
     COPY CMQSMPOV.
**   Property name
 01  NAME
    COPY CMQCHRVV.
**   Property descriptor
 01  PROPDESC.
     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

Parameters used for the PL/I invocation of MQSETMP.

```
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 */
```

System/390 assembler invocation

Parameters used for the System/390 assembler invocation of MQSETMP.

```
        CALL MQSETMP,(HCONN,HMSG,SETMSGHOPTS,NAME,PROPDESC,TYPE,VALUELENGTH,
              VALUE,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN        DS         F     Connection handle
HMSG         DS         D     Message handle
SETMSGOPTS   CMQSMPOA   ,     Options that control the action of MQSETMP
NAME         CMQCHRVA   ,     Property name
PROPDESC     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 STYPE value specified on the call.

Syntax

MQSTAT (HCONN, STYPE, STAT, CMPCOD, REASON)

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 MQCONNX call.

STYPE (10-digit signed integer) – input

Type of status information being requested. The only valid value is:

MQSTAT_TYPE_ASYNC_ERROR
Return information about previous asynchronous put operations.

STS (MQSTS) – input

Status information structure. See "MQSTS – Status reporting structure" on page 270 for details.

CMPCOD (10-digit signed integer) – output

The completion code; it is one of the following:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

REASON (10-digit signed integer) – output

The reason code qualifying *CMPCOD*.

If *CMPCOD* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CMPCOD* 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_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
(2424, X'978') Error with MQSTS structure

MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For detailed information on these codes, see:
- WebSphere MQ Messages

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 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 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 CMPCOD of MQCC_OK and REASON of MQRC_NONE are returned.
3. Counts of the number of asynchronous calls that have been processed under the connection handle are returned via 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, respectively (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.

RPG invocation

```
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
D* Status type
D STYPE                         10I 0
D* Status information structure
D STS                          224A
D* Completion code
D CMPCOD                        10I 0
D* Reason
D REASON                        10I 0
```

MQSUB – Register Subscription

The MQSUB call registers the applications subscription to a particular topic.

Syntax

MQSUB (HCONN, SUBDSC, HOBJ, HSUB, CMPCOD, REASON)

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 MQCONNX call.

On i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

SUBDSC (MQSD) – input/output

This is a structure that identifies the object whose use is being registered by the application. See "MQSD - Subscription Descriptor" on page 248 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 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*, OOOUT (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 this is the case, the call fails with RC2019
- By using the DEFINE SUB MQSC command and providing 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, RC2040, or RC2038 respectively.

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 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 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 summarised 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 wish 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 (10-digit signed integer) – input" on page 428 for information about handle scope. An MQCLOSE of the *HOBJ* handle has no effect on the *HSUB* handle.

HSUB (10-digit signed integer) – input

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 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 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:

RC2298
(2298 X'08FA') Function not supported.

RC2046
(2046 X'07FE') Options not valid or not consistent

RC2019
(2019 X'07E3') Object handle not valid

RC2161
(2161 X'0871') Queue manager quiescing

RC2085
(2085 X'0825') Object identified cannot be found

RC2424
(2424 X'0978') Subscription descriptor (MQSD) not valid

RC2440
(2440 X'0988') SubName field not valid

RC2441
(2441 X'0989') Objectstring field not valid

RC2425
(2441 X'979') Topic string not valid

RC2431
(2431 X'097F') SubUserData field not valid

RC2432
(2432 X'0980') Subscription already exists

RC2434
(2434 X'0982') Subscription name matches existing subscription

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

RC2435
(2435 X'0983') Attribute cannot be changed using SDALT, or subscription was created with SDIMM.

RC2436
(2436 X'0984') MQSO_DURABLE option not valid

RC2503
(2503 X'09C7') MQSUB calls are currently inhibited for the topics subscribed to

Usage notes

1. 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 263.
2. 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 authority for output is set. If the SDMAN option is used, this means 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 an authority check is made on the queue represented by the *HOBJ* parameter.
3. 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.
4. Subscriptions can be grouped together 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 together 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.

5. Fields in the MQSD are filled in 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 subsequently 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, with the exception of 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.
ObjectString	This input only field is unchanged on return from an MQSUB call. The Full topic name used is returned in the *ResObjectString* 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 subsequently 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.

MQSD output from MQSUB

Field name in MQSD	Special considerations
SubUserData and SelectionString	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 *VSBufSize*. If no buffer is provided only the length 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 you subsequently reuse the MQSD in an MQSUB call using the SOALT option and a buffer is not provided but a non-zero *VSLength* is provided, if that length matches the existing length of the field, no alteration will made to the field.
SubCorrelId and PubAccountingToken	If you do not use MQSO_SET_CORREL_ID, then the *SubCorrelId* will be generated by the queue manager. If you do not use MQSO_SET_IDENTITY_CONTEXT, then the *PubAccountingToken* will be generated by the queue manager. 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.
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.

RPG invocation

```
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                        372A
D* Object handle
D HOBJ                           10I 0
D* Subscription handle
D HSUB                           10I 0
D* Completion code
D CMPCOD                         10I 0
D* Reason code qualifying CMPCOD
D REASON                         10I 0
```

MQSUBRQ - Subscription Request

The MQSUBRQ call makes a request on a subscription.

Syntax

MQSUBRQ (HCONN, HSUB, ACTION, SUBROPT, CMPCOD, REASON)

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 MQCONNX call.

On z/OS for CICS applications, and on i5/OS for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

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 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 will be indicated by the MQIsRetained message property of that publication.

Since the topic in the existing subscription represented by the *HSUB* parameter may 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 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.

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 may 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 (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 RC2525 (MQRC_RETAINED_MSG_Q_ERROR) or RC2526 (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 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.

Language invocations

```
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
D* Action requested by MQSUBRQ
D ACTION                        10I 0
D* Options that control the action of MQSUBRQ
D SBROPT                        16A
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0
```

Chapter 3. Attributes of objects

Attributes for queues

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 actually 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 actually placed on the 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 the WebSphere MQ Intercommunication book.
- 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 the WebSphere MQ Intercommunication book.

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 attributes:

Overview

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 Y symbol in Table 86 and subsequent tables.

Table 86 summarizes the attributes that are specific to queues. The attributes are described in alphabetic order.

Note: The names of the attributes shown in this book 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 the WebSphere MQ Script (MQSC) Command Reference for details.

Table 86. 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	Topic
AlterationDate	Date when definition was last changed	Y		Y	Y		AlterationDate
AlterationTime	Time when definition was last changed	Y		Y	Y		AlterationTime
BackoutRequeueQName	Excessive backout requeue queue name	Y	Y				BackoutRequeueQName

Table 86. Attributes for queues (continued). 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	Topic
BackoutThreshold	Backout threshold	Y	Y				BackoutThreshold
BaseQName	Queue name to which alias resolves			Y			BaseQName
ClusterName	Name of cluster to which queue belongs	Y		Y	Y		ClusterName
ClusterNamelist	Name of namelist object containing names of clusters to which queue belongs	Y		Y	Y		ClusterNamelist
CreationDate	Date the queue was created	Y					CreationDate
CreationTime	Time the queue was created	Y					CreationTime
CurrentQDepth	Current queue depth	Y					CurrentQDepth
DefBind	Default binding	Y		Y	Y	Y	DefBind
DefinitionType	Queue definition type	Y	Y				DefinitionType
DefInputOpenOption	Default input open option	Y	Y				DefInputOpenOption
DefPersistence	Default message persistence	Y	Y	Y	Y	Y	DefPersistence
DefPriority	Default message priority	Y	Y	Y	Y	Y	DefPriority
DistLists	Distribution list support	Y	Y				DistLists
HardenGetBackout	Whether to maintain an accurate backout count	Y	Y				HardenGetBackout
InhibitGet	Controls whether get operations for the queue are allowed	Y	Y	Y			InhibitGet

Table 86. Attributes for queues (continued). 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	Topic
InhibitPut	Controls whether put operations for the queue are allowed	Y	Y	Y	Y	Y	InhibitPut
InitiationQName	Name of initiation queue	Y	Y				InitiationQName
MaxMsgLength	Maximum message length in bytes	Y	Y				MaxMsgLength
MaxQDepth	Maximum queue depth	Y	Y				MaxQDepth
MediaLog	Identity of oldest log extent (or oldest journal receiver on i5/OS) needed for media recovery of a specified queue	Y	Y				MediaLog
MsgDeliverySequence	Message delivery sequence	Y	Y				MsgDeliverySequence
OpenInputCount	Number of opens for input	Y					OpenInputCount
OpenOutputCount	Number of opens for output	Y					OpenOutputCount
ProcessName	Process name	Y	Y				ProcessName
QDepthHighEvent	Controls whether Queue Depth High events are generated	Y	Y				QDepthHighEvent
QDepthHighLimit	High limit for queue depth	Y	Y				QDepthHighLimit
QDepthLowEvent	Controls whether Queue Depth Low events are generated	Y	Y				QDepthLowEvent
QDepthLowLimit	Low limit for queue depth	Y	Y				QDepthLowLimit

Table 86. Attributes for queues (continued). 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	Topic
QDepthMaxEvent	Controls whether Queue Full events are generated	Y	Y				QDepthMaxEvent
QDesc	Queue description	Y	Y	Y	Y	Y	QDesc
QName	Queue name	Y		Y	Y	Y	QName
QServiceInterval	Target for queue service interval	Y	Y				QServiceInterval
QServiceIntervalEvent	Controls whether Service Interval High or Service Interval OK events are generated	Y	Y				QServiceIntervalEvent
QType	Queue type	Y		Y	Y	Y	QType
RemoteQMgrName	Name of remote queue manager				Y		RemoteQMgrName
RemoteQName	Name of remote queue				Y		RemoteQName
RetentionInterval	Retention interval	Y	Y				RetentionInterval
Scope	Controls whether an entry for the queue also exists in a cell directory	Y		Y	Y		Scope
Shareability	Queue shareability	Y	Y				Shareability
TriggerControl	Trigger control	Y	Y				TriggerControl
TriggerData	Trigger data	Y	Y				TriggerData
TriggerDepth	Trigger depth	Y	Y				TriggerDepth
TriggerMsgPriority	Threshold message priority for triggers	Y	Y				TriggerMsgPriority
TriggerType	Trigger type	Y	Y				TriggerType
Usage	Queue usage	Y	Y				Usage
XmitQName	Transmission queue name				Y		XmitQName

AlterationDate (12-byte character string)

Date when definition was last changed.

Local	Model	Alias	Remote	Cluster
Y		Y	Y	

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, `1992-09-23bb`, where `bb` represents 2 blank characters).

It is normal for the values of certain attributes to change as the queue manager operates (for example, *CurrentQDepth*). Changes to these attributes do not affect *AlterationDate*. Also, changes resulting from use of the MQSET call 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
Y		Y	Y	

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.

It is normal for the values of certain attributes to change as the queue manager operates (for example, *CurrentQDepth*). Changes to these attributes do not affect *AlterationTime*. Also, changes resulting from use of the MQSET call 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
Y	Y			

Apart from allowing its value to be queried, the queue manager takes no action based on the value of this 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
Y	Y			

Apart from allowing its value to be queried, the queue manager takes no action based on the value of this 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
		Y		

This is the name of a queue that is defined to the local queue manager. (For more information on 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.

CFStrucName (12-byte character string)

Coupling-facility structure name.

Local	Model	Alias	Remote	Cluster
Y	Y			

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.

ClusterName (48-byte character string)

Name of cluster to which queue belongs.

Local	Model	Alias	Remote	Cluster
Y		Y	Y	

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
Y		Y	Y	

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
Y				

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, `1992-09-23bb`, where `bb` represents 2 blank characters).

- On i5/OS, the creation date of a queue may 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
Y				

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 i5/OS, the creation time of a queue may 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 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
Y				

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 125.

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
Y		Y	Y	Y

This is the default binding that is used when OOBNDQ is specified on the MQOPEN call and the queue is a cluster queue. The value is one of the following:

BNDOPN
Binding fixed by MQOPEN call.

BNDNOT
Binding not fixed.

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
Y	Y			

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 471).

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 313 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 313 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
Y	Y			

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. The value is one of the following:

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).

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.

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
Y	Y	Y	Y	Y

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)

The value is one of the following:

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
Y	Y	Y	Y	Y

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 437 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 471.

To determine the value of this attribute, use the IADPRI selector with the MQINQ call.

DistLists (10-digit signed integer)

Distribution list support.

Local	Model	Alias	Remote	Cluster
Y	Y			

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 is capable of processing 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).

The value is one of the following:

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
Y	Y			

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. This count is available in the *MDBOC* 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 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 may or may not be incremented.

Hardening information for each MQGET call within a unit of work, however, imposes a performance overhead, and the *HardenGetBackout* attribute should be set to QABH only if it is essential that the count is accurate.

- On i5/OS, 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 may not be remembered.

Hardening is not used to ensure that the backout count for messages on this queue is accurate. The count may 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
Y	Y	Y		

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 subsequently 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
Y	Y	Y	Y	Y

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. The value is one of the following:

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 subsequently 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
Y	Y			

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 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 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
Y	Y			

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 usually resource constraints imposed by the operating system, or by the environment in which the application is running, will result 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 to too big for the queue
- RC2031 if the message to 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 determined by the environment:

- On i5/OS, 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 395.

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
Y	Y			

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 i5/OS) needed for media recovery of a particular queue.

Local	Model	Alias	Remote	Cluster
Y	Y			

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
Y	Y			

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
Y				

This is the number of handles that are currently valid for removing messages from the queue by means of 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 action(s) 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
Y				

This is the number of handles that are currently valid for adding messages to the queue by means of 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 action(s) 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
Y	Y			

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
Y	Y			

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:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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
Y	Y			

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
Y	Y			

A Queue Depth Low 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 *QDepthLowLimit* attribute).

Note: The value of this attribute can change dynamically.

The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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
Y	Y			

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
Y	Y			

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.

The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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
Y	Y	Y	Y	Y

This is a field that may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may 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 may 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
Y		Y	Y	Y

This is the name of a queue defined on the local queue manager. For more information about queue names, see the WebSphere MQ Application Programming Guide. All queues defined on a queue manager share the same queue name space. 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
Y	Y			

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
Y	Y			

- 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:

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 the Monitoring WebSphere MQ book.

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
Y		Y	Y	

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
Y		Y	Y	Y

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
			Y	

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 whose name 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.

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 whose name 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 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
			Y	

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
Y	Y			

This is the period of time for 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 and time of the queue are recorded in the *CreationDate* and *CreationTime* attributes, respectively.

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 whose retention interval 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
Y		Y	Y	

A cell directory is provided by an installable Name service. The value is one of the following:

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 WebSphere 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 IASCOP selector with the MQINQ call.

Support for this attribute is subject to the following restrictions:

- On i5/OS, 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
Y	Y			

This indicates whether the queue can be opened for input multiple times concurrently. The value is one of the following:

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
Y	Y			

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
Y	Y			

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
Y	Y			

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
Y	Y			

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 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 471); 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
Y	Y			

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.

TTFRST
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
Y	Y			

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 the WebSphere MQ Application Programming Guide.

Only privileged applications can open a transmission queue for OOOUT 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 286), otherwise errors may 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
			Y	

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 whose name 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 CAXQN selector with the MQINQ call. The length of this attribute is given by LNQN.

Attributes for namelists

The following table summarizes the attributes that are specific to namelists. The attributes are described in alphabetic order.

Note: The names of the attributes shown in this book 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 the WebSphere MQ Script (MQSC) Command Reference for details.

Table 87. Attributes for namelists

Attribute	Description	Topic
AlterationDate	Date when definition was last changed	AlterationDate
AlterationTime	Time when definition was last changed	AlterationTime
NameCount	Number of names in namelist	NameCount
NamelistDesc	Namelist description	NamelistDesc
NamelistName	Namelist name	NamelistName
Names	A list of *NameCount* names	Names

Attribute descriptions

A namelist object has the attributes described below.

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 may 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 may 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 may 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. For more information about namelist names, see the WebSphere MQ Application Programming Guide.

Each namelist has a name that is different from the names of other namelists belonging to the queue manager, but may 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×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 the WebSphere MQ Application Programming Guide.

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

The following table summarizes the attributes that are specific to process definitions. The attributes are described in alphabetic order.

Note: The names of the attributes shown in this book 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 the WebSphere MQ Script (MQSC) Command Reference for details.

Table 88. Attributes for process definitions

Attribute	Description	Topic
AlterationDate	Date when definition was last changed	AlterationDate
AlterationTime	Time when definition was last changed	AlterationTime
ApplId	Application identifier	ApplId
ApplType	Application type	ApplType
EnvData	Environment data	EnvData
ProcessDesc	Process description	ProcessDesc
ProcessName	Process name	ProcessName
UserData	User data	UserData

Attribute descriptions

A process-definition object has the attributes described below.

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 WebSphere 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, but the following values are recommended for standard types; user-defined application types should be restricted to values in the range ATUFST through ATULST:

ATCICS
CICS transaction.

AT400 i5/OS 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 WebSphere 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 may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may 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 may 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 may 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 WebSphere MQ simply 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

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 alphabetic order.

Note: The names of the attributes shown in this book 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 the WebSphere MQ Script (MQSC) Command Reference for details.

Table 89. Attributes for the queue manager

Attribute	Description	Topic
AlterationDate	Date when definition was last changed	AlterationDate
AlterationTime	Time when definition was last changed	AlterationTime
AuthorityEvent	Controls whether authorization (Not Authorized) events are generated	AuthorityEvent
BridgeEvent	Controls whether IMS bridge events are generated.	BridgeEvent
ChannelAutoDef	Controls whether automatic channel definition is permitted	ChannelAutoDef
ChannelAutoDefEvent	Controls whether channel automatic-definition events are generated	ChannelAutoDefEvent
ChannelAutoDefExit	Name of user exit for automatic channel definition	ChannelAutoDefExit
ChannelEvent	Controls whether channel events are generated	ChannelEvent
ClusterCacheType	Controls whether the cluster cache is fixed in size or dynamically sized	ClusterCacheType
ClusterWorkloadData	User data for cluster workload exit	ClusterWorkloadData
ClusterWorkloadExit	Name of user exit for cluster workload management	ClusterWorkloadExit
ClusterWorkloadLength	Maximum length of message data passed to cluster workload exit	ClusterWorkloadLength
CodedCharSetId	Coded character set identifier	CodedCharSetId
CommandEvent	Controls whether command event messages are queued	CommandEvent
CommandInputQName	Command input queue name	CommandInputQName
CommandLevel	Command level	CommandLevel
ConfigurationEvent	Configuration event	ConfigurationEvent
DeadLetterQName	Name of dead-letter queue	DeadLetterQName
DefXmitQName	Default transmission queue name	DefXmitQName
DistLists	Distribution list support	DistLists
InhibitEvent	Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated	InhibitEvent
LocalEvent	Controls whether local error events are generated	LocalEvent
LoggerEvent	Controls whether recovery log events are generated	LoggerEvent
MaxHandles	Maximum number of handles	MaxHandles
MaxMsgLength	Maximum message length in bytes	MaxMsgLength
MaxPriority	Maximum priority	MaxPriority
MaxUncommittedMsgs	Maximum number of uncommitted messages within a unit of work	MaxUncommittedMsgs
PerformanceEvent	Controls whether performance-related events are generated	PerformanceEvent
Platform	Platform on which the queue manager is running	Platform
QMgrDesc	Queue manager description	QMgrDesc
QMgrIdentifier	Unique internally-generated identifier of queue manager	QMgrIdentifier
QMgrName	Queue manager name	QMgrName
QPubSub	Whether the queued publish/subscribe interface is running	QPubSub

Table 89. Attributes for the queue manager (continued)

Attribute	Description	Topic
RemoteEvent	Controls whether remote error events are generated	RemoteEvent
RepositoryName	Name of cluster for which this queue manager provides repository services	RepositoryName
RepositoryNamelist	Name of namelist object containing names of clusters for which this queue manager provides repository services	RepositoryNamelist
SQQMName	Controls whether or not the messages in a queue-sharing group are put onto a locally-defined shared queue, or directly onto the target queue.	
SSLCRLNamelist	Name of namelist object containing names of authentication information objects.	Note 1
SSLEvent	Controls whether SSL events are generated	SSLEvent
SSLKeyRepository	Location of SSL key repository.	Note 1
SSLKeyResetCount	Determines the number of non-encrypted bytes sent and received within an SSL conversation before the encryption key is renegotiated.	SSLKeyResetCount
StartStopEvent	Controls whether start and stop events are generated	StartStopEvent
SyncPoint	Syncpoint availability	SyncPoint
TraceRouteRecording	Controls the recording of trace route information for messages	TraceRouteRecording
TriggerInterval	Trigger-message interval	TriggerInterval

Notes:

1. This attribute cannot be inquired using the MQINQ call, and is not described in this book. See the WebSphere MQ Programmable Command Formats and Administration Interface book for details of this attribute.

Attribute descriptions

The queue manager object has the attributes described below.

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 value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

To determine the value of this attribute, use the IAAUTE selector with the MQINQ call.

BridgeEvent (character string)

Determines whether or not IMS bridge event messages are generated.

This attribute determines whether IMS bridge event messages are put onto the SYSTEM.ADMIN.CHANNEL.EVENT queue. This attribute is only supported on z/OS.

The value can be one of the following:

- MQEVR_ENABLED (MQINQ/config event) ENABLED (MQSC): The following IMS Bridge events are generated: MQRC_BRIDGE_STARTED, MQRC_BRIDGE_STOPPED. This value is only supported on z/OS.
- MQEVR_DISABLED (MQINQ/config event) DISABLED (MQSC): IMS Bridge events are not generated. This is the queue manager's initial default value.

To determine the value of this attribute, use the MQIA_IMS_BRIDGE_EVENT selector with the MQINQ call.

ChannelAutoDef (10-digit signed integer)

Controls whether automatic channel definition is permitted.

This attribute controls the automatic definition of channels of type CTRCVR and CTSVCN. Note that the automatic definition of CTCLSD channels is always enabled. The value is one of the following:

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 CTRCVR, CTSVCN, and CTCLSD. The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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 CTRCVR, 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 LNEXN.

ChannelEvent (character string)

Determines whether or not 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'). Prior to 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 channel events off, 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 'interesting' 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:

- MQEVR_EXCEPTION (only the following channel events are generated: MQRC_CHANNEL_ACTIVATED, MQRC_CHANNEL_CONV_ERROR, MQRC_CHANNEL_NOT_ACTIVATED, MQRC_CHANNEL_STOPPED, CHANNEL_STOPPED_BY_USER).
- MQEVR_ENABLED (all channel events are generated; that is, in addition to the events generated by MQEVR_EXCEPTION, the MQRC_CHANNEL_STARTED, and MQRC_CHANNEL_STOPPED events are also generated).
- MQEVR_DISABLED (no channel events are generated; this is the queue manager initial default value).

To determine the value of this attribute, use the MQIA_CHANNEL_EVENT 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 nonblank, 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 decide whether to accept the queue instance selected by the queue manager as the destination for the message, or choose 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 LNEXN.

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, including the names of objects, queue creation date and

time, and so on. 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 i5/OS, 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 or not 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 MQIA_COMMAND_EVENT with one of the following values:

- MQEVR_ENABLED — command event messages are generated and put onto the queue for all successful commands.
- MQEVR_NO_DISPLAY — 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.
- MQEVR_DISABLED — 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 CMDEVselector with the MQINQ call.

CommandInputQName (48-byte character string)

Command input queue name.

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 i5/OS, 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. Refer to the WebSphere MQ Programmable Command Formats and Administration Interface. book for details of the Escape command.

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 value is one of the following:

CMLVL1
Level 1 of system control commands.

This value is returned by the following:

- MQSeries for OS/400

- Version 2 Release 3
- Version 3 Release 1
- Version 3 Release 6

CML320
Level 320 of system control commands.

This value is returned by the following:

- MQSeries for OS/400
  - Version 3 Release 2
  - Version 3 Release 7

CML420
Level 420 of system control commands.

This value is returned by the following:

- MQSeries for AS/400®
  - Version 4 Release 2.0
  - Version 4 Release 2.1

CML510
Level 510 of system control commands.

This value is returned by the following:

- MQSeries for AS/400 Version 5 Release 1

CML520
Level 520 of system control commands.

This value is returned by the following:

- MQSeries for AS/400 Version 5 Release 2

CML530
Level 530 of system control commands.

This value is returned by the following:

- WebSphere MQ for i5/OS Version 5 Release 3

CML600
Level 600 of system control commands.

This value is returned by the following:

- WebSphere MQ for i5/OS Version 6 Release 0

CML700
Level 700 of system control commands.

This value is returned by the following:

- WebSphere MQ for i5/OS Version 7 Release 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:

- MQEVR_ENABLED
- MQEVR_DISABLED.

If the ConfigurationEvent attribute is set to MQEVR_ENABLED, and certain commands are successfully issued via 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, QSVCIEV.
- 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 (usually 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 125) 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 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 71 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 WebSphere 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.

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. The value is one of the following:

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.

The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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 the Monitoring WebSphere MQ book.

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.

The value is one of the following:

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 the Monitoring WebSphere MQ book.

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 i5/OS, 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 usually 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 i5/OS, 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 may 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.

The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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 i5/OS.

QMgrDesc (64-byte character string)

Queue manager description.

This is a field that may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may 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 may be translated incorrectly if this field is sent to another queue manager.

On i5/OS, 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 125). 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.

QPubSub (10-digit signed integer)

Whether the queued publish/subscribe interface is running and hence getting from SYSTEM.BROKER.CONTROL.QUEUE and the queues listed in SYSTEM.QPUBSUB.QUEUE.NAMELIST.

The value is one of the following:

STOPPED
The queued publish/subscribe interface is not running.

RUNNING
The queued publish/subscribe interface is running and getting from the queues listed above.

To determine the value of this attribute, use the MQIA_QUEUED_PUBSUB selector with the MQINQ call.

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 the Monitoring WebSphere MQ book.

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 LNQMN.

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.

SQQMName (character string)

This determines whether or not the messages in a queue-sharing group are put onto a locally-defined shared queue, or directly onto the target queue.

The value is one of the following:

- USE: messages are delivered to the ObjectQMgrName before being put onto the target queue. This is the default value. It ensures that when a queue manager is migrated from a previous version the function remains unchanged.
- IGNORE: messages are put directly onto the shared queue by any queue manager in the same queue-sharing group as the ObjectQMgrName.

By putting a given message directly onto its target queue in a queue sharing group, WebSphere MQ does not have two execute two puts and a get. This feature decreases the workload in a queue sharing group (the decrease in workload could be even bigger, where messages are being transmitted over channels).

To determine the value of this attribute, use the MQIA_SQQMNAME selector with the MQINQ call.

SSLEvent (character string)

Determines whether or not SSL events are generated.

The value is one of the following:

- MQEVR_ENABLED (MQINQ/PCF/config event) ENABLED (MQSC): SSL events are generated (that is, the MQRC_CHANNEL_SSL_ERROR event is generated).
- MQEVR_DISABLED (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 MQIA_SSL_EVENT 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 may 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 between 1 byte and 32Kb, SSL/TLS channels will use a secret key reset count of 32Kb. This is to avoid the overhead of excessive key resets which would occur for small SSL/TLS secret key reset values.

When the SSL server is a WebSphere 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 MQIA_SSL_RESET_COUNT selector with the MQINQ call.

StartStopEvent (10-digit signed integer)

Controls whether start and stop events are generated.

The value is one of the following:

EVRDIS
Event reporting disabled.

EVRENA
Event reporting enabled.

For more information about events, see the Monitoring WebSphere MQ book.

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)

Trace route recording.

This controls whether or not information about messages is recorded as they flow through a queue manager. The value is one of the following:

- DISABLED: no appending to trace route messages is allowed
- MQROUTE_RECORDING_Q: messages are put onto a fixed named queue
- MQROUTE_RECORDING_MSG: 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 MQRO_DISCARD_MSG report option. To prevent report or reply messages remaining in the system, set the report option MQRO_PASS_DISCARD_AND_EXPIRY. For more information, see Chapter 8, "Report options and message flags," on page 517.

To determine the value of this attribute, use the IATRGI 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 on triggering, see the WebSphere MQ Application Programming Guide.

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.

Attributes for authentication information

The following table summarizes the attributes that are specific to authentication information objects. The attributes are described in alphabetic order.

Note: The names of the attributes shown in this book 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 the WebSphere MQ Script (MQSC) Command Reference for details.

Table 90. Attributes for process definitions

Attribute	Description	Topic
AlterationDate	Date when definition was last changed	AlterationDate
AlterationTime	Time when definition was last changed	AlterationTime

Table 90. Attributes for process definitions (continued)

Attribute	Description	Topic
AuthInfoConnName	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.	AuthInfoConnName
AuthInfoDesc	Plain-text comment. It provides descriptive information about the authentication information object when an operator issues the DISPLAY AUTHINFO command.	AuthInfoDesc
AuthInfoName	Name of the authentication information object.	AuthInfoName
AuthInfoType	The type of authentication information.	AuthInfoType
LDAPPassword	The password associated with the Distinguished Name of the user who is accessing the LDAP server.	LDAPPassword
LDAPUserName	The Distinguished Name of the user who is accessing the LDAP server.	LDAPUserName

Attribute descriptions

An authentication information object has the attributes described below.

AlterationDate (MQCHAR12)

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.

AlterationTime (MQCHAR8)

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` 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.

AuthInfoConnName (MQCHAR264)

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.

The syntax for CONNAME is the same as for channels. For example,
`conname('hostname(nnn)')`

where *nnn* is the port number. If *nnn* is not provided, the default port number 389 is used.

The maximum length for the field is 264 characters.

AuthInfoDesc (MQCHAR64)

Plain-text comment. It provides descriptive information about the authentication information object when an operator issues the DISPLAY AUTHINFO command.

It should contain only displayable characters. The maximum length is 64 characters. In a DBCS installation, it can contain DBCS characters (subject to a maximum length of 64 bytes).

Note: If characters are used that are not in the coded character set identifier (CCSID) for this queue manager, they might be translated incorrectly if the information is sent to another queue manager.

AuthInfoName (MQCHAR48)

Name of the authentication information object.

The name must not be the same as any other authentication information object name currently defined on this queue manager (unless REPLACE or ALTER is specified).

AuthInfoType (MQLONG)

The type of authentication information. The value must be CRLLDAP, meaning that Certificate Revocation List checking is done using LDAP servers.

LDAPPassword (MQCHAR32)

The password associated with the Distinguished Name of the user who is accessing the LDAP server.

Its maximum size is 32 characters. The default value is blank.

LDAPUserName (MQ_DISTINGUISHED_NAME_LENGTH)

The Distinguished Name of the user who is accessing the LDAP server.

The maximum size for the user name is 1024 characters on i5/OS, UNIX systems, and Windows, and 256 characters on z/OS.

The maximum accepted line length is defined to be BUFSIZ, which can be found in stdio.h.

If you use asterisks (*) in the user name they are treated as literal characters, and not as wild cards, because LDAPUSER is a specific name and not a string used for matching.

Chapter 4. Applications

Building your application

The i5/OS publications describe how to build executable applications from the programs you write. This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building WebSphere MQ for i5/OS applications to run under i5/OS.

In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the WebSphere MQ for i5/OS 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.

WebSphere MQ copy files

WebSphere MQ for i5/OS 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 WebSphere MQ for i5/OS provides to assist with the writing of channel exits are described in the WebSphere MQ Intercommunication book.

The names of the WebSphere MQ for i5/OS 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 Table 2 on page 6.

Note: For ILE RPG/400® they are supplied as members of file QRPGLESRC 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 WebSphere MQ for i5/OS application, you have to compile the source code you have written.

To do this for ILE RPG/400, you can use the usual i5/OS commands, CRTRPGMOD and CRTPGM.

After creating your *MODULE, you need to specify `BNDSRVPGM(QMQM/LIBMQM)` in the CRTPGM command. This includes the various WebSphere 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.

Interfaces to the i5/OS external syncpoint manager

WebSphere MQ for i5/OS uses native i5/OS commitment control as an external syncpoint coordinator. See the *i5/OS Programming: Backup and Recovery Guide* Guide for more information about the commitment control capabilities of i5/OS.

To start the i5/OS 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 WebSphere MQ for i5/OS. For example:

```
STRCMTCTL LCKLVL(*ALL) CMTSCOPE(*JOB)
```

If you call MQPUT, MQPUT1, or MQGET, specifying PMSYP or GMSYP, after starting commitment control, WebSphere MQ for i5/OS 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.

WebSphere MQ for i5/OS 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 ,WebSphere MQ for i5/OS remains registered as an API commitment resource so that it is notified of the next commit or rollback. When the next syncpoint is reached, WebSphere 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 WebSphere 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 i5/OS commitment control is used as an external syncpoint coordinator, MQCMIT, MQBACK, and MQBEGIN calls may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

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 i5/OS applications

WebSphere MQ for i5/OS 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 WebSphere MQ for i5/OS 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 WebSphere MQ for i5/OS 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

This chapter describes the sample programs delivered with WebSphere MQ for i5/OS 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 91 gives a complete list of the sample programs delivered with WebSphere MQ for i5/OS, 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 91. Names of the sample programs

	RPG (ILE)
Put samples	AMQ3PUT4
Browse samples	AMQ3GBR4
Get samples	AMQ3GET4

Table 91. Names of the sample programs (continued)

	RPG (ILE)
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 WebSphere MQ for i5/OS 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 the WebSphere MQ for i5/OS System Administration Guide. You could use the sample CL program to create queues to use with the sample programs described in this chapter.

For information on how to run the sample programs, see "Preparing and running the sample programs" on page 495.

Features demonstrated in the sample programs

Table 92 shows the techniques demonstrated by the WebSphere MQ for i5/OS 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.

Table 92. Sample programs demonstrating use of the MQI

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

Table 92. Sample programs demonstrating use of the MQI (continued)

Technique	RPG (ILE)
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

Before you can run the WebSphere MQ for i5/OS sample programs, you must compile them as you would any other WebSphere MQ for i5/OS applications. To do this, you can use the i5/OS commands CRTRPGMOD and CRTPGM.

When you create the `AMQ3xxx4` programs, you need to specify BNDSRVPGM(QMQM/LIBMQM) in the CRTPGM command. This includes the various MQ procedures in your program.

The sample programs are provided in library QMQMSAMP as members of QRPGLESRC. They use the copy files provided in library QMQM, so make sure 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.

Note that 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 the WebSphere MQ for i5/OS System Administration Guide book) to do this, or edit and run AMQSAMP4 with the alternative definition.

The Put sample program

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

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 OOBRW 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; at this point 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

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 (*GMWI*) 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

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.

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 CHGMQMPRC command (described in the WebSphere MQ for i5/OS System Administration Guide) to do this, or edit and run your own version of AMQSAMP4.

3. You need to 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 1 on page 501. 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, respectively:

  ```
  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.

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 OOOUT 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 (*GMWI*) 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 93 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.

Table 93. Client/Server sample program details

Program name	SYSTEM/SAMPLE queue	Program started
Echo	ECHO	AMQ3ECH4
Inquire	INQ	AMQ3INQ4
Set	SET	AMQ3SET4

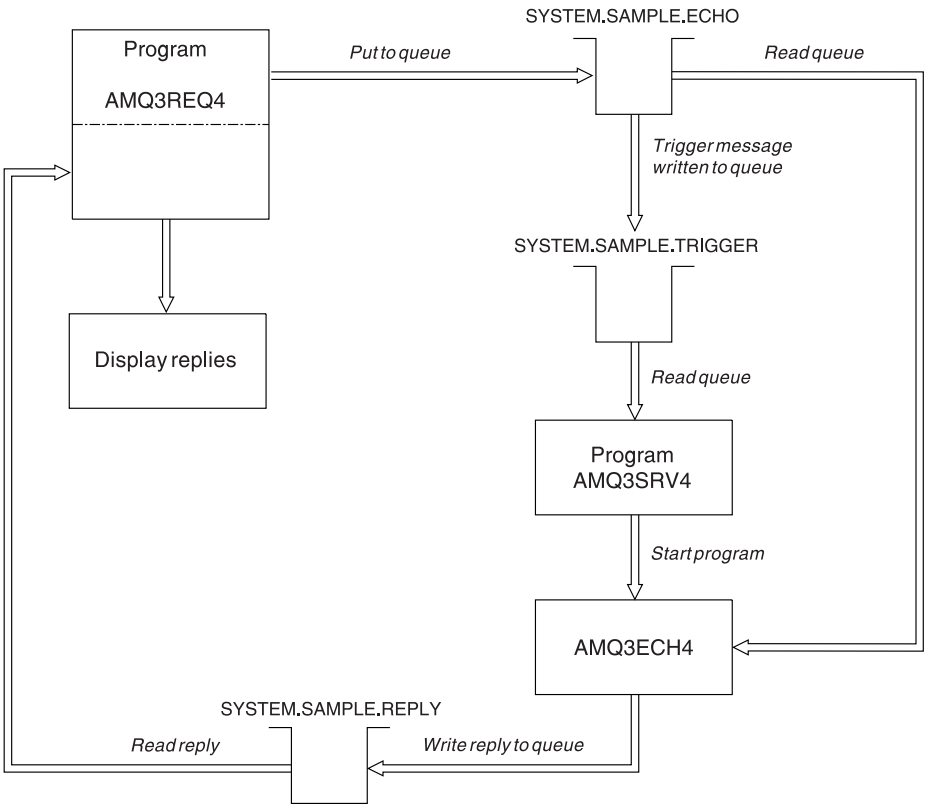

Figure 1. Sample Client/Server (Echo) program flowchart

The Echo sample program

The Echo sample programs return the message send to a reply queue. The program is named AMQ3ECH4

The programs are intended to run as triggered programs, so their only input is the data read from the queue named in the trigger message structure.

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 CHGMQMPRC command, described in the WebSphere MQ for i5/OS System Administration Guide.) 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 WebSphere MQ for i5/OS, 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 WebSphere MQ for i5/OS, 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 WebSphere MQ for i5/OS sending channel, if the program you have written does not provide similar conversion for the reply.

The Inquire sample program

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 that contains the name of a target queue whose attributes are to be inquired.

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 this, specify the name of the Inquire sample program in the *ApplId* field of the SYSTEM.SAMPLE.INQPROCESS process definition. (For this, you can use the CHGMQMPRC command, described in the WebSphere MQ for i5/OS System Administration Guide book.) 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 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.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 this is not necessary for WebSphere MQ for i5/OS, 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 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 3 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

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 whose attributes are to be inquired.

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 CHGMQMPRC command, described in the WebSphere MQ for i5/OS System Administration Guide.) 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 WebSphere MQ for i5/OS, 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 OOSET 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

WebSphere MQ for i5/OS supplies two Triggering sample programs that are written in ILE/RPG. The programs are:

AMQ3TRG4
: This is a trigger monitor for the i5/OS environment. It submits an i5/OS job for the application to be started, but this means there is a processing overhead associated with each trigger message.

AMQ3SRV4
: This is a trigger server for the i5/OS 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

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 i5/OS 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 i5/OS 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 similar to that of the trigger monitor, except the trigger server:

- Allows CICS as well as i5/OS applications
- Does not use the environment data from the trigger message
- Calls i5/OS applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an i5/OS 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

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:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*NO)
```

Note: To start triggering again on this queue, you *must* enter the command:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*YES)
```

Running the samples using remote queues

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 on how to do this, see the WebSphere MQ Intercommunication book.

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.

Chapter 5. Return codes for i5/OS (ILE RPG)

This section describes the return codes associated with the MQI and MQAI.

The return codes associated with:

- Programmable Command Format (PCF) commands are listed in WebSphere MQ Programmable Command Formats and Administration Interface.
- C++ calls are listed in WebSphere MQ 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 i5/OS (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 may 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 may be a fault in the application program, or it may be a result of some situation external to the program, for example the user's authority may have been revoked. The *REASON* parameter gives additional information about the error.

Reason codes

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 WebSphere MQ Messages.

To find your i5/OS 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 94.

i5/OS	**Other platforms**
CCOK	MQCC_OK
CCWARN	MQCC_WARN
CCFAIL	MQCC_FAIL

Chapter 6. Rules for validating MQI options

This appendix lists the situations that produce an RC2046 reason code from an MQOPEN, MQPUT, MQPUT1, MQGET, or MQCLOSE call.

MQOPEN call

For the options of the MQOPEN call:

- *At least one* of the following must be specified:
  - OOBRW
  - OOINPQ
  - OOINPX
  - OOINPS
  - OOINQ
  - OOOUT
  - OOSET
- Only *one* of the following is allowed:
  - OOINPQ
  - OOINPX
  - OOINPS
- Only *one* of the following is allowed:
  - OOBNDO
  - OOBNDN
  - OOBNDQ

  Note: The options listed above 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 OOSET* or OOPAS* options is specified, OOOUT must also be specified.

MQPUT call

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

For the put-message options, the rules are the same as for the MQPUT call, except for the following:

- PMALTU is allowed.
- PMLOGO is *not* allowed.

MQGET call

For the get-message options:

- Only *one* of the following is allowed:
  - GMNSYP
  - GMSYP
  - GMPSYP
- Only *one* of the following is allowed:
  - GMBRWF
  - GMBRWC
  - GMBRWN
  - GMMUC
- GMSYP is not allowed with any of the following:
  - GMBRWF
  - GMBRWC
  - GMBRWN
  - GMLK
  - GMUNLK
- GMPSYP is not allowed with any of the following:
  - GMBRWF
  - GMBRWC
  - GMBRWN
  - GMCMPM
  - GMUNLK
- If GMLK is specified, one of the following must also be specified:
  - GMBRWF
  - GMBRWC
  - GMBRWN
- If GMUNLK is specified, only the following are allowed:
  - GMNSYP
  - GMNWT

MQCLOSE call

- 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

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:
  - MQSO_ALTER
  - MQSO_RESUME
  - MQSO_CREATE
- Only one of the following is allowed:
  - MQSO_DURABLE
  - MQSO_NON_DURABLE

  Note: The options listed above are mutually exclusive. However, as the value of MQSO_NON_DURABLE is 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
- The combination of MQSO_NEW_PUBLICATIONS_ONLY and MQSO_PUBLICATIONS_ON_REQUEST is not allowed.
- MQSO_NEW_PUBLICATIONS_ONLY is only allowed in combination with MQSO_CREATE.
- Only one of the following is allowed:
  - MQSO_WILDCARD_CHAR
  - MQSO_WILDCARD_TOPIC

Chapter 7. Machine encodings

This appendix describes the structure of the *MDENC* field in the message descriptor (see "MQMD – Message descriptor" on page 125).

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

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 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.

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

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 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 which 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.

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

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 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 may 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 together. 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 below.

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

Encodings for machine architectures are shown in Table 95.

Table 95. Summary of encodings for machine architectures

Machine architecture	Binary integer encoding	Packed-decimal integer encoding	Floating-point encoding
i5/OS	normal	normal	IEEE normal
Intel® x86	reversed	reversed	IEEE reversed
PowerPC®	normal	normal	IEEE normal
System/390	normal	normal	System/390

Chapter 8. Report options and message flags

This appendix concerns the *MDREP* and *MDMFL* fields that are part of the message descriptor MQMD specified on the MQGET, MQPUT, and MQPUT1 calls (see "MQMD – Message descriptor" on page 125). The appendix 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 above 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 desired 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, applications that need to check whether the sender of the message requested a particular report should use the technique described below.

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 = MQRO_EXCEPTION
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 above, but replacing the value 8 in the steps above 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 above are returned.

Chapter 9. Data conversion

This appendix 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."
- 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 525.
- Special considerations for the conversion of report messages; see "Conversion of report messages" on page 529.
- The parameters passed to the data-conversion exit; see "MQCONVX - Data conversion exit" on page 542.
- A call that can be used from the exit in order to convert character data between different representations; see "MQXCNVC - Convert characters" on page 536.
- The data-structure parameter which is specific to the exit; see "MQDXP – Data-conversion exit parameter" on page 530.

Conversion processing

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, prior to 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 usually 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. See below for possible reason codes.

Processing conventions

When converting a built-in format, the queue manager follows the processing conventions described below. It is recommended that 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:

FMADMN
FMCICS
FMCMD1
FMCMD2
FMDLH
FMDH
FMEVNT
FMIMS
FMIMVS
FMMDE
FMPCF
FMRMH
FMRFH
FMRFH2
FMSTR
FMTM
FMXQH

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 may 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, 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 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 or not 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 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 may 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 may or may 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. With the exception of 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 actually returned to the application; the values returned for such fields within the message data are the values applicable to the message *prior to 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 above 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 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 *MDFMT*, *MDCSI*, and *MDENC* fields in the MQMD structure contained *within* the MQXQH. For each subsequent 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 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 desired for the application message data, prior to 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 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 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 above is 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 may therefore imply a length of data which is different from the length actually present in the report message (cases 1 and 2 above).

If the GMCONV option is specified when the report message is retrieved:

- For case 1 above, the data-conversion exit will not be invoked (because the report message will have no data).
- For case 3 above, the format name correctly implies the length of the message data.
- But for case 2 above, 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 usually 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 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

The following table summarizes the fields in the structure.

Table 96. Fields in MQDXP

Field	Description	Topic
DXSID	Structure identifier	DXSID
DXVER	Structure version number	DXVER
DXAOP	Application options	DXAOP
DXENC	Numeric encoding required by application	DXENC
DXCSI	Character set required by application	DXCSI
DXLEN	Length in bytes of message data	DXLEN
DXCC	Completion code	DXCC
DXREA	Reason code qualifying *DXCC*	DXREA
DXRES	Response from exit	DXRES
DXHCN	Connection handle	DXHCN

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 may 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

The MQDXP structure contains the following fields; the fields are described in **alphabetic 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 recommendations 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 actually 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 then 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 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 listed below, 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 listed below 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 wishes 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
```

MQXCNVC - Convert characters

The MQXCNVC call converts characters from one character set to another.

This call is part of the WebSphere MQ Data Conversion Interface (DCI), which is one of the WebSphere MQ framework interfaces. Note: this call can be used only from a data-conversion exit.

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 i5/OS, 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 below can be specified. If more than one is required, the values can be added together (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 *SRCLEN*. On return from MQXCNVC, *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 neither added nor 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 big 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 *SRCLEN*. On return from MQXCNVC, *DATLEN* is less than or equal to *TGTLEN*.

Encoding options: The options described below 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 whose representation in main storage 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 above 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 515 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 whose representation 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 above 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*.

SRCLEN (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×SRCLEN) – 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 *SRCLEN*.

TGTBUF (1-byte character string×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 big 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 on these reason codes, see Chapter 5, "Return codes for i5/OS (ILE RPG)," on page 507.

RPG invocation (ILE)

```
C*..1....:....2....:....3....:....4....:....5....:....6....:....7..
C                    CALLP     MQXCNVC(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..
DMQXCNVC          PR                  EXTPROC('MQXCNVC')
D* Connection handle
D HCONN                          10I 0 VALUE
D* Options that control the action of MQXCNVC
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 actually provided by the queue manager (see usage note 11 on page 545).

This definition is part of the WebSphere MQ Data Conversion Interface (DCI), which is one of the WebSphere MQ framework interfaces.

Syntax

MQCONVX *(MQDXP, MQMD, INLEN, INBUF, OUTLEN, OUTBUF)*

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 530 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 i5/OS, 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 prior to 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×INLEN) – input

Buffer containing the unconverted message.

This contains the message data prior to 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×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.

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 529 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 big 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* and *DXREA* fields set to CCWARN and RC2110 respectively.

   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 has the effect of returning 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 may 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 similar to 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 actually 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.

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

Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information 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 information. The furnishing of this information does not give 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:

IBM World Trade Asia Corporation,
Licensing,
2-31 Roppongi 3-chome, Minato-k,u
Tokyo 106, 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 information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information 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 United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

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 Programming License Agreement, or any equivalent agreement between us.

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.

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. 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 IBM's application programming interfaces.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	CICS
CICS/VSE	COBOL/400	FFST
IBM	IBMLink	Informix
i5/OS	IMS	Lotus Notes
MQSeries	MVS	OS/2
OS/390	OS/400	PowerPC
RACF	RPG/400	S/390
System/390	VSE/ESA	WebSphere
z/OS		

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Unix is a trademark of The Open Group in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44-1962-816151
  - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

SC34-6943-00

Spine information:

WebSphere MQ for i5/OS Application Programming Reference (ILE RPG) Version 7.0