MQSeries

Using C++

SC33-1877-01

MQSeries

Using C++

SC33-1877-01

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix D, “Notices” on page 109.

Second edition (February 1998)

This edition applies to the following products:

- MQSeries for AIX Version 5
- MQSeries for AS/400 Version 4 Release 2
- MQSeries for HP-UX Version 5
- MQSeries for OS/2 Warp Version 5
- MQSeries for Sun Solaris Version 5
- MQSeries for Windows NT Version 5

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

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page titled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories,
Information Development,
Mail Point 095,
Hursley Park,
Winchester,
Hampshire,
England,
SO21 2JN

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

Contents

Figures

Tables

About this book

This publication describes the C++ programming-language binding to the Message Queue Interface (MQI). This part of the MQSeries products is referred to as *MQSeries C++*.

MQSeries C++ is supplied as part of the following products:

- MQSeries for AIX Version 5
- MQSeries for AS/400 V4R2
- MQSeries for HP-UX Version 5
- MQSeries for OS/2 Warp Version 5
- MQSeries for Sun Solaris Version 5
- MQSeries for Windows NT Version 5

The information is intended for application programmers who write programs to make use of the MQI.

What you need to know

You should have:

- Knowledge of the C programming language
- Knowledge of the C++ programming language
- Understanding of the purpose of the Message Queue Interface (MQI) as described in the *MQSeries Application Programming Guide*, and the *MQSeries Application Programming Reference*
- Experience of MQSeries programs in general, or familiarity with the content of the other MQSeries publications

How to use this book

First read Chapter 1, “Introduction to MQSeries C++” on page 1. This is a programming guide as well as an introduction.

There are some things specific to C++ that you may need to know in Chapter 2, “C++ language considerations” on page 17.

The main, reference part of the book is Chapter 3, “MQSeries C++ classes” on page 21.

The Appendixes contain information about compiling and linking your programs, a cross reference to the MQSeries data structures, object attributes, and calls, and some additional reason codes.

MQSeries publications

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries “family” books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX V5.0
- MQSeries for AS/400 V4R2
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Digital OpenVMS V2.2
- MQSeries for HP-UX V5.0
- MQSeries for MVS/ESA V1.2
- MQSeries for OS/2 Warp V5.0
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for SunOS V2.2
- MQSeries for Sun Solaris V5.0
- MQSeries for Tandem NonStop Kernel V2.2
- MQSeries Three Tier
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.0

Any exceptions to this general rule are indicated. (Publications that support the MQSeries Level 1 products are listed in “MQSeries Level 1 product publications” on page ix. For a functional comparison of the Level 1 and Level 2 MQSeries products, see the *MQSeries Planning Guide*.)

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
MQSeries: An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, the dead-letter queue handler, and the MQSeries links for Lotus Notes**. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0

MQSeries Command Reference
The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management
The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, programmable command formats (PCFs), and installable services.

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes “AMQ” messages issued by MQSeries, applies to these MQSeries products only:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

MQSeries Application Programming Guide
The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference
The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary
The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++
MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by V5.0 of MQSeries for AIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, and by MQSeries clients supplied with those products and installed in the following environments:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95

| MQSeries C++ is also supported by MQSeries for AS/400 V4R2.

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX V5.0 Quick Beginnings, GC33-1867

| **MQSeries for AS/400**

| *MQSeries for AS/400 Version 4 Release 2 Licensed Program Specifications*,
| GC33-1958

| *MQSeries for AS/400 Version 4 Release 2 Administration Guide*, GC33-1956

| *MQSeries for AS/400 Version 4 Release 2 Application Programming Reference*
| *(RPG)*, SC33-1957

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX Version 2.2 System Management Guide, SC33-1642

MQSeries for Digital OpenVMS

MQSeries for Digital OpenVMS Version 2.2 System Management Guide, GC33-1791

MQSeries for HP-UX

MQSeries for HP-UX V5.0 Quick Beginnings, GC33-1869

MQSeries for MVS/ESA

MQSeries for MVS/ESA Version 1 Release 2 Licensed Program Specifications, GC33-1350

MQSeries for MVS/ESA Version 1 Release 2 Program Directory

MQSeries for MVS/ESA Version 1 Release 2 System Management Guide, SC33-0806

MQSeries for MVS/ESA Version 1 Release 2 Messages and Codes, GC33-0819

MQSeries for MVS/ESA Version 1 Release 2 Problem Determination Guide, GC33-0808

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp V5.0 Quick Beginnings, GC33-1868

MQSeries link for R/3

MQSeries link for R/3 Version 1.0 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx Version 2.2 System Management Guide, GC33-1768

MQSeries for SunOS

MQSeries for SunOS Version 2.2 System Management Guide, GC33-1772

MQSeries for Sun Solaris

MQSeries for Sun Solaris V5.0 Quick Beginnings, GC33-1870

| **MQSeries for Tandem NonStop Kernel**

| *MQSeries for Tandem NonStop Kernel Version 2.2 System Management*
| *Guide*, GC33-1893

MQSeries Three Tier

MQSeries Three Tier Administration Guide, SC33-1451
MQSeries Three Tier Reference Summary, SX33-6098
MQSeries Three Tier Application Design, SC33-1636
MQSeries Three Tier Application Programming, SC33-1452

MQSeries for Windows

MQSeries for Windows Version 2.0 User's Guide, GC33-1822

MQSeries for Windows Version 2.1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT V5.0 Quick Beginnings, GC33-1871

MQSeries Level 1 product publications

For information about the MQSeries Level 1 products, see the following publications:

MQSeries: Concepts and Architecture, GC33-1141

MQSeries Version 1 Products for UNIX Operating Systems Messages and Codes, SC33-1754

MQSeries for SCO UNIX Version 1.4 User's Guide, SC33-1378

MQSeries for UnixWare Version 1.4.1 User's Guide, SC33-1379

MQSeries for VSE/ESA Version 1 Release 4 Licensed Program Specifications, GC33-1483

MQSeries for VSE/ESA Version 1 Release 4 User's Guide, SC33-1142

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries products, including all MQSeries V5.0 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

HTML format

The MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0

The MQSeries books are also available from the MQSeries product family Web site:

| `http://www.software.ibm.com/ts/mqseries/`

Information Presentation Facility (IPF) format

In the OS/2 environment, the MQSeries documentation is supplied in IBM IPF format on the MQSeries product CD-ROM.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

MQSeries web site

The MQSeries product family Web site is at:

```
http://www.software.ibm.com/ts/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML format.
- Download MQSeries SupportPacs.

Related publications

The Booch methodology
Object-Oriented Analysis and Design with Applications 2nd Edition, by Grady Booch, Benjamin/Cummings Publishing, ISBN 0-8053-5340-2.

Summary of changes

Changes to the previous edition are marked in the left-hand margin with bars.

Changes for this edition

MQSeries C++ is now supplied as part of MQSeries for AS/400 Version 4 Release 2, in addition to being supplied as part of the MQSeries Version 5 products.

Chapter 1. Introduction to MQSeries C++

MQSeries C++ allows you to write MQSeries application programs in the C++ programming language.

This chapter introduces the features of MQSeries C++. There are details about preparing message data, reading messages and writing messages to the dead-letter queue. The sample programs provided are introduced and there is a sample program listing. Implicit operations (connect, open, reopen, close and disconnect) are explained and there are some notes about binary and character strings.

MQSeries C++ can be used with the following products when they have been installed as a full queue manager (the MQSeries *Base product and server*):

- MQSeries for AIX Version 5
- | MQSeries for AS/400 Version 4 Release 2
- MQSeries for HP-UX Version 5
- MQSeries for OS/2 Warp Version 5
- MQSeries for Sun Solaris Version 5
- MQSeries for Windows NT Version 5

MQSeries C++ can also be used with an MQSeries client supplied with the above
| Version 5 products and installed on the following platforms:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows 3.1
- Windows 95
- Windows NT

Features of MQSeries C++

MQSeries C++ provides the following features:

- Automatic initialization of MQSeries data structures
- Just-in-time queue manager connection and queue opening
- Implicit queue closure and queue manager disconnection
- Dead-letter header transmission and receipt
- IMS Bridge header transmission and receipt
- Reference message header transmission and receipt
- Trigger message receipt

All the classes in the following Booch class diagrams broadly parallel those MQSeries entities in the procedural MQI (for example C) that have either handles or data structures. All classes inherit from the ImqError (see "ImqError" on page 33) class, which allows an error condition to be associated with each object.

In the Booch methodology, each class is identified by a name within a cloud. Below the class name may be listed any noteworthy attributes and methods. An abstract class is denoted by a small triangle within a cloud. Inheritance is denoted by an arrow to the parent class. A cooperative relationship between two classes is denoted by an undecorated line between clouds. A referential relationship between two classes is denoted by a line decorated with numbers indicating the number of objects that may participate in a given relationship at any one time.

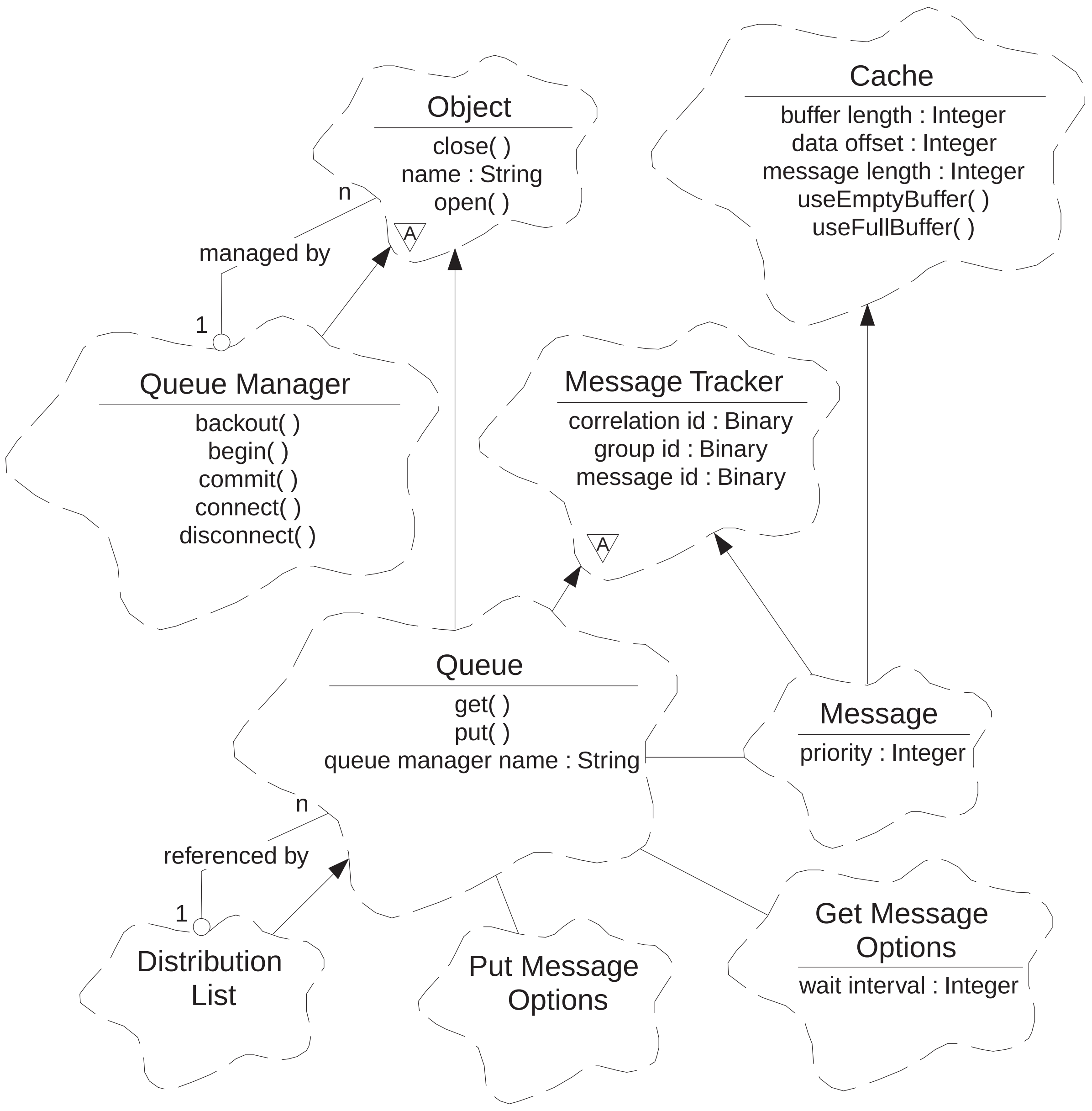


Figure 1. MQSeries C++ classes (queue management)

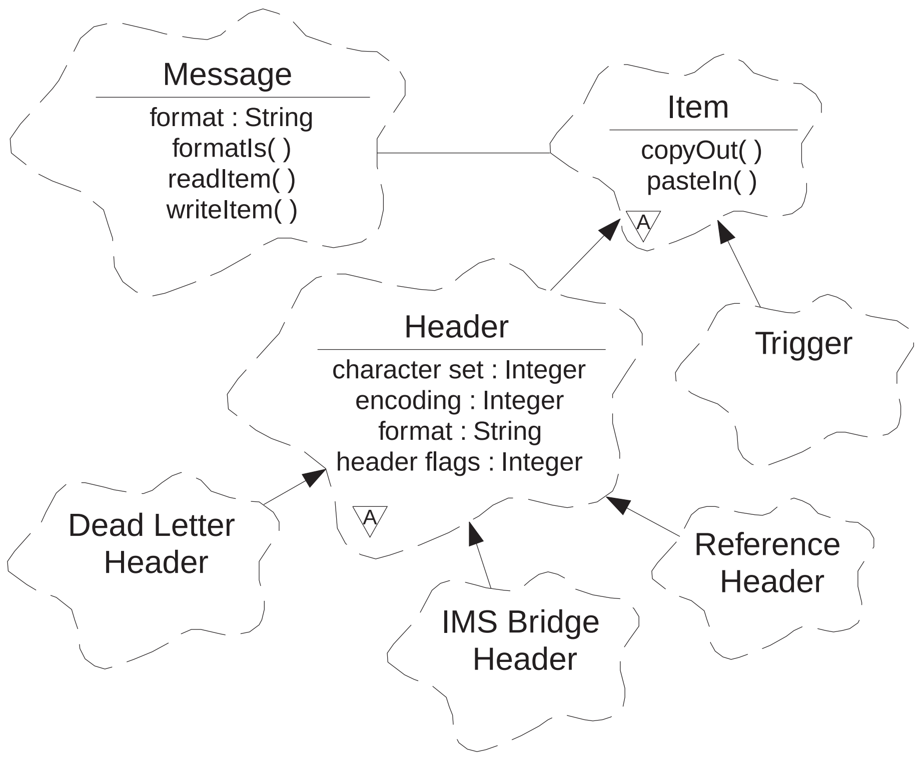


Figure 2. MQSeries C++ classes (item handling)

The following classes and data types are used in the C++ method signatures of the queue management classes (see Figure 1 on page 2) and the item handling classes (see Figure 2):

- The ImqBinary class (see "ImqBinary" on page 23) which encapsulates byte arrays such as MQBYTE24.
- The **ImqBoolean** data type which is defined as **typedef unsigned char ImqBoolean**.
- The ImqString class (see "ImqString" on page 82) which encapsulates character arrays such as MQCHAR64.

Entities with data structures are subsumed within appropriate object classes. Individual data structure fields (see Appendix B, "MQI cross-reference" on page 95) are accessed with methods.

Entities with handles come under the ImqObject (see "ImqObject" on page 53) class hierarchy and provide encapsulated interfaces to the MQI. Objects of these classes exhibit intelligent behavior that can reduce the number of method invocations required relative to the procedural MQI. For example, you can establish and discard queue manager connections as required, or you can open a queue with appropriate options, then close it.

The ImqMessage class (see "ImqMessage" on page 45) encapsulates the MQMD data structure and also acts as a holding point for user data and *items* (see "Reading messages" on page 5) by providing cached buffer facilities. You can provide fixed-length buffers for user data and use the buffer many times, the amount of data present in the buffer can vary from one use to the next. Alternatively, the system can provide and manage a buffer of flexible length. Both the size of the buffer (the amount available for receipt of messages) and the amount actually used (either the number of bytes for transmission or the number of bytes actually received) become important considerations.

Preparing message data

When you send a message, message data is first prepared in a buffer managed by an ImqCache object (see "ImqCache" on page 25). A buffer is associated (by inheritance) with each ImqMessage object (see "ImqMessage" on page 45): it can be supplied by the application (using either the **useEmptyBuffer** or **useFullBuffer** method); or it can be supplied automatically by the system. The advantage of the application supplying the message buffer is that no data copying is necessary in many cases because the application can use prepared data areas directly; the disadvantage is that the supplied buffer is of a fixed length.

The buffer can be reused, and the number of bytes transmitted can be varied each time if desired by using the **setMessageLength** method prior to transmission.

When supplied automatically by the system, the number of bytes available is managed by the system, and data can be copied into the message buffer using, for example, the ImqCache **write** method, or the ImqMessage **writeItem** method. The message buffer grows according to need. As the buffer grows, there is no loss of previously written data. A large or multi-part message can be written in sequential pieces.

The following fragments show simplified straightforward message sends:

```
/* 1. Use prepared data in a user-supplied buffer. */
char pszBuffer[ ] = "Hello world" ;

msg.useFullBuffer( pszBuffer, sizeof( pszBuffer ) );
msg.setFormat( MQFMT_STRING );

/* 2. Use prepared data in a user-supplied buffer, */
/* where the buffer size exceeds the data size. */
char pszBuffer[ 24 ] = "Hello world" ;

msg.useEmptyBuffer( pszBuffer, sizeof( pszBuffer ) );
msg.setFormat( MQFMT_STRING );
msg.setMessageLength( 12 );

/* 3. Copy data to a user-supplied buffer. */
char pszBuffer[ 12 ];

msg.useEmptyBuffer( pszBuffer, sizeof( pszBuffer ) );
msg.setFormat( MQFMT_STRING );
msg.write( 12, "Hello world" );

/* 4. Copy data to a system-supplied buffer. */
msg.setFormat( MQFMT_STRING );
msg.write( 12, "Hello world" );

/* 5. Copy data to a system-supplied buffer using objects. */
/* (Objects set the message format as well as content.) */
ImqString strText( "Hello world" );

msg.writeItem( strText );
```

Figure 3. Ways of preparing message data

Reading messages

When receiving data, the application or the system can supply a suitable message buffer. The same buffer can be used for both multiple transmission and multiple receipt for a given ImqMessage object. If the message buffer is supplied automatically, it grows to accommodate whatever length of data is received. However, if the application supplies the message buffer, it might not be big enough. Then either truncation or failure might occur, depending on the options used for message receipt.

Incoming data can be accessed directly from the message buffer, in which case the data length indicates the total amount of incoming data. Alternatively, incoming data can be read sequentially from the message buffer. In this case, the data pointer addresses the next byte of incoming data, and the data pointer and data length are updated each time data is read.

Items are pieces of a message, all in the user area of the message buffer, that need to be processed sequentially and separately. Apart from regular user data, an item might be a dead-letter header or a trigger message. Items are always associated with message formats; message formats are ***not*** always associated with items.

There is a class of object for each item that corresponds to a recognizable MQSeries message format. There is one for a dead-letter header and one for a trigger message. There is no class of object for user data. That is, once the recognizable formats have been exhausted, processing the remainder is left to the application program. Classes for user data can be written by specializing the ImqItem class.

This next fragment shows a message receipt that takes account of all potential items that can precede the user data, in an imaginary situation. Non-item user data is simply defined as anything that occurs after items that can be identified. An automatic buffer (the default) is used to hold an arbitrary amount of message data.

```
ImqQueue queue ;
ImqMessage msg ;

if ( queue.get( msg ) ) {

  /* Process all items of data in the message buffer. */
  do while ( msg.dataLength( ) ) {
    ImqBoolean bFormatKnown = FALSE ;

    /* There remains unprocessed data in the message buffer. */

    /* Determine what kind of item is next. */

    if ( msg.formatIs( MQFMT_DEAD_LETTER_HEADER ) ) {
      ImqDeadLetterHeader header ;

      /* The next item is a dead-letter header.             */
      /* For the next statement to work and return TRUE,     */
      /* the correct class of object pointer must be supplied. */
      bFormatKnown = TRUE ;

      if ( msg.readItem( header ) ) {

        /* The dead-letter header has been extricated from the */
        /* buffer and transformed into a dead-letter object.   */
        /* The encoding and character set of the dead-letter   */
        /* object itself are MQENC_NATIVE and MQCCSI_Q_MGR.    */
        /* The encoding and character set from the dead-letter */
        /* header have been copied to the message attributes   */
        /* to reflect any remaining data in the buffer.        */

        /* Process the information in the dead-letter object.  */
        /* Note that the encoding and character set have       */
        /* already been processed.                             */
        ...
      }
```

Figure 4 (Part 1 of 3). Retrieving items within a message

```
      /* There might be another item after this, */
      /* or just the user data.                  */
    }

    if ( msg.formatIs( MQFMT_TRIGGER ) ) {
      ImqTrigger trigger ;

      /* The next item is a trigger message.                 */
      /* For the next statement to work and return TRUE,     */
      /* the correct class of object pointer must be supplied. */
      bFormatKnown = TRUE ;

      if ( msg.readItem( trigger ) ) {

        /* The trigger message has been extricated from the */
        /* buffer and transformed into a trigger object.    */

        /* Process the information in the trigger object. */
        ...
      }

      /* There is usually nothing after a trigger message. */
    }

    if ( msg.formatIs( FMT_USERCLASS ) ) {
      UserClass object ;

      /* The next item is an item of a user-defined class.     */
      /* For the next statement to work and return TRUE,       */
      /* the correct class of object pointer must be supplied. */
      bFormatKnown = TRUE ;

      if ( msg.readItem( object ) ) {

        /* The user-defined data has been extricated from the */
        /* buffer and transformed into a user-defined object. */

        /* Process the information in the user-defined object. */
        ...
      }

      /* Continue looking for further items. */
    }

    if ( ! bFormatKnown ) {
      /* There remains data which is not associated with a specific   */
      /* item class.                                                  */
      char * pszDataPointer = msg.dataPointer( );       /* Address. */
      int iDataLength = msg.dataLength( );              /* Length.  */
```

Figure 4 (Part 2 of 3). Retrieving items within a message

```
      /* The encoding and character set for the remaining data are  */
      /* reflected in the attributes of the message object, even    */
      /* if a dead-letter header was present.                       */
      ...

    }

  }
}
```

Figure 4 (Part 3 of 3). Retrieving items within a message

With an automatic buffer, it is important to remember that the buffer storage is volatile. That is, buffer data might be held at a different physical location after each **get** method invocation. Therefore each time buffer data is referenced, use the **bufferPointer** or **dataPointer** methods to access message data.

You may want a program to set aside a fixed area for receiving message data. In this case invoke the **useEmptyBuffer** method before using the **get** method.

Using a fixed, non-automatic area limits messages to a maximum size, so it is important to consider the MQGMO_ACCEPT_TRUNCATED_MSG option of the ImqGetMessageOptions object. If this option is not specified (this is the default), the MQRC_TRUNCATED_MSG_FAILED reason code can be expected. If this option is specified, the MQRC_TRUNCATED_MSG_ACCEPTED reason code may be expected depending upon the design of the application.

This next code fragment shows how a fixed area of storage might be used to receive messages:

```
char * pszBuffer = new char[ 100 ];

msg.useEmptyBuffer( pszBuffer, 100 );
gmo.setOptions( MQGMO_ACCEPT_TRUNCATED_MSG );
queue.get( msg, gmo );

delete [ ] pszBuffer ;
```

Figure 5. Retrieving messages into a fixed area of storage

Note: The responsibility for discarding a user-defined (non-automatic) buffer rests with the application, not with the ImqCache class object.

In the above fragment, the buffer can always be addressed directly, with *pszBuffer*, as opposed to using the **bufferPointer** method, although it is advisable to use the **dataPointer** method for general-purpose access.

Note: Specifying a null pointer and zero length with **useEmptyBuffer** does not nominate a fixed length buffer of length zero, as might be expected. This combination is actually interpreted as a request to ignore any previous user-defined buffer, and instead revert to the use of an automatic buffer.

Writing a message to the dead-letter queue

A typical case of a multi-part message is one containing a dead-letter header. The data from a message that cannot be processed is appended to the dead-letter header.

```
ImqQueueManager mgr ;        // The queue manager.
ImqQueue queueIn ;           // Incoming message queue.
ImqQueue queueDead ;         // Dead-letter message queue.
ImqMessage msg ;             // Incoming and outgoing message.
ImqDeadLetterHeader header ; // Dead-letter header information.

// Retrieve the message to be rerouted.
queueIn.setConnectionReference( mgr );
queueIn.setName( MY_QUEUE );
queueIn.get( msg );

// Set up the dead-letter header information.
header.setDestinationQueueManagerName( mgr.name( ) );
header.setDestinationQueueName( queueIn.name( ) );
header.setPutApplicationName( /* ? */ );
header.setPutApplicationType( /* ? */ );
header.setPutDate( /* TODAY */ );
header.setPutTime( /* NOW */ );
header.setDeadLetterReasonCode( FB_APPL_ERROR_1234 );

// Insert the dead-letter header information. This will vary
// the encoding, character set and format of the message.
// Message data is moved along, past the header.
msg.writeItem( header );

// Send the message to the dead-letter queue.
queueDead.setConnectionReference( mgr );
queueDead.setName( mgr.deadLetterQueueName( ) );
queueDead.put( msg );
```

Figure 6. Writing a message to the dead-letter queue

Writing a message to the IMS bridge

Messages sent to MQSeries for MVS/ESA via the IMS bridge require a special header. The IMS bridge header is prefixed to regular message data.

```
ImqQueueManager mgr ;       // The queue manager.
ImqQueue queueIn ;          // Incoming message queue.
ImqQueue queueBridge ;      // IMS bridge message queue.
ImqMessage msg ;            // Incoming and outgoing message.
ImqImsBridgeHeader header ; // IMS bridge header information.


// Retrieve the message to be forwarded.
queueIn.setConnectionReference( mgr );
queueIn.setName( MY_QUEUE );
queueIn.get( msg );


// Set up the IMS bridge header information.
// The reply-to format is often specified.
// Other attributes can be specified, but all have default values.
header.setReplyToFormat( /* ? */ );

// Insert the IMS bridge header information. This will vary
// the encoding, character set and format of the message.
// Message data is moved along, past the header.
msg.writeItem( header );

// Send the message to the IMS bridge queue.
queueBridge.setConnectionReference( mgr );
queueBridge.setName( /* ? */  );
queueBridge.put( msg );
```

Figure 7. Writing a message to the IMS bridge

The sample programs

The sample programs are:

- HELLO WORLD (imqwrld.cpp)
- SPUT (imqsput.cpp) and SGET (imqsget.cpp)
- DPUT (imqdput.cpp)

Sample program HELLO WORLD (imqwrld.cpp)

This program shows how to put or get a regular datagram (C structure) using the ImqMessage class. This sample employs few method invocations, taking advantage of implicit method invocations such as **open**, **close**, and **disconnect**.

Using a server connection to MQSeries:

- Run[1] **imqwrlds** to use the existing default queue SYSTEM.DEFAULT.LOCAL.QUEUE.

[1] For details of executing AS/400 programs see “Compiling C++ sample programs for the AS/400” on page 92

- Run **imqwrlds** SYSTEM.DEFAULT.MODEL.QUEUE to use a temporary dynamically assigned queue.

Using a client connection to MQSeries:

- Run **imqwrldc**.

```
extern "C" {
#include <stdio.h>
}

#include <imqi.hpp> // MQSeries C++

#define EXISTING_QUEUE "SYSTEM.DEFAULT.LOCAL.QUEUE"

#define BUFFER_SIZE 12

static char gpszHello[ BUFFER_SIZE ] = "Hello world" ;

int main ( int argc, char * * argv ) {
  ImqQueueManager manager ;
  int iReturnCode = 0 ;

  // Connect to the queue manager.
  if ( argc > 2 ) {
    pmanager -> setName( argv[ 2 ] );
  }
  if ( pmanager -> connect( ) ) {
    ImqQueue * pqueue = new ImqQueue ;
    ImqMessage * pmsg = new ImqMessage ;

    // Identify the queue which will hold the message.
    pqueue -> setConnectionReference( manager );
    if ( argc > 1 ) {
      pqueue -> setName( argv[ 1 ] );

      // The named queue can be a model queue, which will result in the
      // creation of a temporary dynamic queue, which will be destroyed
      // as soon as it is closed. Therefore we must ensure that such a
      // queue is not automatically closed and reopened. We do this by
      // setting open options which will avoid the need for closure and
      // reopening.
      pqueue -> setOpenOptions( MQOO_OUTPUT | MQOO_INPUT_SHARED |
                                MQOO_INQUIRE );
```

Figure 8 (Part 1 of 3). The HELLO WORLD sample program

```
    } else {
      pqueue -> setName( EXISTING_QUEUE );

      // The existing queue is not a model queue, and will not be
      // destroyed by automatic closure and reopening. Therefore we will
      // let the open options be selected on an as-needed basis. The
      // queue will be opened implicitly with an output option during
      // the "put", and then implicitly closed and reopened with the
      // addition of an input option during the "get".
    }

    // Prepare a message containing the text "Hello world".
    pmsg -> useFullBuffer( gpszHello , BUFFER_SIZE );
    pmsg -> setFormat( MQFMT_STRING );

    // Place the message on the queue, using default put message options.
    // The queue will be automatically opened with an output option.
    if ( pqueue -> put( * pmsg ) ) {
      ImqString strQueue( pqueue -> name( ) );

      // Discover the name of the queue manager.
      ImqString strQueueManagerName( manager.name( ) );
      printf( "The queue manager name is %s.\n",
              (char *)strQueueManagerName );

      // Show the name of the queue.
      printf( "Message sent to %s.\n", (char *)strQueue );


      // Retrieve the data message just sent ("Hello world" expected)
      // from the queue, using default get message options. The queue
      // is automatically closed and reopened with an input option
      // if it is not already open with an input option. We get the
      // message just sent, rather than any other message on the
      // queue, because the "put" will have set the ID of the message
      // so, as we are using the same message object, the message ID
      // acts as in the message object, a filter which says that we
      // are interested in a message only if it has this particular ID.
      if ( pqueue -> get( * pmsg ) ) {
        int iDataLength = pmsg -> dataLength( );
```

Figure 8 (Part 2 of 3). The HELLO WORLD sample program

```
        // Show the text of the received message.
        printf( "Message of length %d received, ", iDataLength );

        if ( pmsg -> formatIs( MQFMT_STRING ) ) {
          char * pszText = pmsg -> bufferPointer( );

          // If the last character of data is a null, then we can
          // assume that the data can be interpreted as a text string.
          if ( ! pszText[ iDataLength - 1 ] ) {
            printf( "text is \"%s\".\n", pszText );
          } else {
            printf( "no text.\n" );
          }

        } else {
          printf( "non-text message.\n" );
        }

      } else {
        printf( "ImqQueue::get failed with reason code %ld\n",
                pqueue -> reasonCode( ) );
        iReturnCode = (int)pqueue -> reasonCode( );
      }

    } else {
      printf( "ImqQueue::open/put failed with reason code %ld\n",
              pqueue -> reasonCode( ) );
      iReturnCode = (int)pqueue -> reasonCode( );
    }

    // Deletion of the queue will ensure that it is closed.
    // If the queue is dynamic then it will also be destroyed.
    delete pqueue ;
    delete pmsg ;

  } else {
    printf( "ImqQueueManager::connect failed with reason code %ld\n",
            manager.reasonCode( ) );
    iReturnCode = (int)manager.reasonCode( );
  }

  // Destruction of the queue manager ensures that it is
  // disconnected.  If the queue object were still available
  // and open (which it is not), the queue would be closed
  // prior to disconnection.

  return iReturnCode ;
}
```

Figure 8 (Part 3 of 3). The HELLO WORLD sample program

Sample programs SPUT (imqsput.cpp) and SGET (imqsget.cpp)

These programs place messages to and retrieve messages from a named queue.

1. Run **imqsputs** *queue-name*
2. Type in lines at the console, which are placed with MQSeries as messages.
3. Enter a null line to end the input.
4. Run **imqsgets** *queue-name* to retrieve all the lines and display them at the console.

These samples show the use of the following classes:

ImqError (see "ImqError" on page 33)
ImqMessage (see "ImqMessage" on page 45)
ImqObject (see "ImqObject" on page 53)
ImqQueue (see "ImqQueue" on page 62)
ImqQueueManager (see "ImqQueueManager" on page 73)

Sample program DPUT (imqdput.cpp)

This is a distribution list program that puts messages to a distribution list consisting of two queues.

1. Run **imqdputs** *queue-name-1 queue-name-2* to place messages on the two named queues.
2. Run **imqsgets** *queue-name-1* and **imqsgets** *queue-name-2* to retrieve the messages from those queues.

DPUT shows the use of class ImqDistributionList (see "ImqDistributionList" on page 31).

Implicit operations

Several operations can occur implicitly, "just in time" to satisfy the prerequisite conditions for the successful execution of a method. These implicit operations are connect, open, reopen, close, and disconnect.

Connect

An ImqQueueManager object is connected automatically for any method that results in any call to the MQI (see Appendix B, "MQI cross-reference" on page 95).

Open

An ImqObject object is opened automatically for any method that results in an MQGET, MQINQ, MQPUT or MQSET call. The **openFor** method is used to specify one or more relevant **open option** values.

Reopen

An ImqObject is reopened automatically for any method that results in an MQGET, MQINQ, MQPUT or MQSET call, where the object is already open, but the existing **open options** are not adequate to allow the MQI call to be successful. The object is temporarily closed using a temporary **close options** value of MQCO_NONE. The **openFor** method is used to add a relevant **open option**.

Reopen can cause problems in specific circumstances.

- A temporary dynamic queue is destroyed when it is closed and can never be reopened.
- A queue opened for exclusive input (either explicitly or by default) might be accessed by others in the window of opportunity during closure and reopening.
- A browse cursor position is lost when a queue is closed. This situation will not prevent closure and reopening, but will prevent subsequent use of the cursor until MQGMO_BROWSE_FIRST is used again.
- The context of the last message retrieved is lost when a queue is closed.

If any of these circumstances occur or can be foreseen, then avoid reopens by explicitly setting adequate **open options** before an object is opened (either explicitly or implicitly).

Setting the **open options** explicitly for complex queue-handling situations results in better performance and avoids the potential problems listed above.

Close

An ImqObject is closed automatically at any point where the object state would no longer be viable, for example if an ImqObject **connection reference** is severed, or if an ImqObject object is destroyed.

Disconnect

An ImqQueueManager is disconnected automatically at any point where the connection would no longer be viable, for example if an ImqObject **connection reference** is severed, or if an ImqQueueManager object is destroyed.

Binary and character strings

Methods that set character (**char ***) data always take a copy of the data, but some methods might truncate the copy, because certain limits are imposed by MQSeries.

The ImqString class (see "ImqString" on page 82) encapsulates the traditional **char *** and provides support for:

- Comparison
- Concatenation
- Copying
- Integer-to-text and text-to-integer conversion
- Token (word) extraction
- Uppercase translation

The ImqBinary class (see "ImqBinary" on page 23) encapsulates binary byte arrays of arbitrary size, but in particular it is used to hold these attributes:

accounting token (MQBYTE32)

correlation id (MQBYTE24)

group id (MQBYTE24)

instance id (MQBYTE24)

message id (MQBYTE24)

transaction instance id (MQBYTE16)

of objects of these classes:

ImqImsBridgeHeader (see “ImqImsBridgeHeader” on page 40)

ImqMessageTracker (see “ImqMessageTracker” on page 50)

ImqReferenceHeader (see “ImqReferenceHeader” on page 79)

and provides support for:

- Comparison
- Copying

Chapter 2. C++ language considerations

This chapter details the aspects of the C++ language that you must consider when writing application programs that use the Message Queue Interface (MQI).

Header files

Header files are provided as part of the definition of the MQI, to assist with the writing of MQSeries application programs in the C++ language. These header files are summarized in the following table.

Table 1. C/C++ header files

Filename	**Contents**
IMQI.HPP	C++ MQI Classes (includes CMQC.H and IMQTYPE.H)
IMQTYPE.H	Defines the **ImqBoolean** data type
CMQC.H	MQI data structures and manifest constants

To improve the portability of applications, it is recommended that the name of the header file should be coded in lowercase on the **#include** preprocessor directive:

```
#include <imqi.hpp> // C++ classes
```

Methods

Parameters that are const are *input only*. Parameters whose signature includes a pointer (*) or a reference (&) are passed by reference. Return values that do not include a pointer or a reference are passed by value; in the case of returned objects these are new entities that become the responsibility of the caller.

Some method signatures include items that take a default if not specified. Such items are always at the end of signatures and are denoted by an equality sign (=); the value after the equality sign indicates the default value that applies if the item is omitted.

All methods are mixed case beginning with lowercase. Each word except the first within a method name begins with a capital letter. Abbreviations are not used unless their meaning is widely understood. Abbreviations used include “id” for identity and also “sync” for synchronization.

Attributes

Object attributes are accessed using “set” and “get” methods. A “set” method begins with the word “set” whereas a “get” method has no prefix. If an attribute is *read only* there is no “set” method.

Attributes are initialized to valid states during object construction, and the state of an object is always consistent.

Data types

All data types are defined by the C **typedef** statement. The type **ImqBoolean** is defined as **unsigned character** in IMQTYPE.H and can have the values TRUE and FALSE. You can use **ImqBinary** class objects in place of **MQBYTE** arrays, and **ImqString** class objects in place of **char ***. Many methods return objects rather than **char** or **MQBYTE** pointers to ease storage management. All return values become the responsibility of the caller, and in the case of a returned object the storage can be easily disposed of using delete.

Elementary data types

Table 2. Elementary data types

Data Type	Representation
ImqBoolean	**typedef unsigned char ImqBoolean** ;

Manipulating binary strings

Strings of binary data are declared as objects of the **ImqBinary** class. Objects of this class may be copied, compared, and set using the familiar C operators. For example:

```
#include <imqi.hpp> // C++ classes

ImqMessage message ;
ImqBinary id, correlationId ;
MQBYTE24 byteId ;

correlationId.set( byteId, sizeof( byteId ) ); // Set.
id = message.id( );                            // Assign.
if ( correlationId == id ) {                   // Compare.
  ...
```

Figure 9. Manipulating binary strings

Manipulating character strings

When character data is accepted or returned using C++ methods, the character data is always null-terminated and may be of any length. However, certain limits are imposed by MQSeries which may result in information being truncated. To ease storage management, character data is often returned in **ImqString** class objects. These objects can be cast to **char *** and used for *read only* purposes in many situations where a **char *** is required.

Note: The **char *** in an **ImqString** class object may be null.

Although C functions may be used on the **char ***, there are special methods of the **ImqString** class which are preferable; **operator length**( ) is the equivalent of **strlen** and **storage**( ) indicates the memory allocated for the character data.

Initial state of objects

All objects have a consistent initial state reflected by their attributes. The initial values are defined in the class descriptions.

Using C from C++

When using C functions from a C++ program, include headers as in the following example:

```
extern "C" {
#include <string.h>
}
```

Notational conventions

This shows how the methods should be invoked and how the parameters should be declared:

ImqBoolean ImqQueue::**get**(**ImqMessage &** *msg*)

Declare and use the parameters as follows:

```
ImqQueueManager * pmanager ;    // Queue manager
ImqQueue * pqueue ;             // Message queue
ImqMessage msg ;                // Message
char pszBuffer[ 100 ];          // Buffer for message data

pmanager = new ImqQueueManager ;
pqueue = new ImqQueue ;
pqueue -> setName( "myreplyq" );
pqueue -> setConnectionReference( pmanager );

msg.useEmptyBuffer( pszBuffer, sizeof( pszBuffer ) );

if ( pqueue -> get( msg ) ) {
  long lDataLength = msg.dataLength( );

  ...
}
```

Figure 10. Declaration and use conventions

Chapter 3. MQSeries C++ classes

This library component encapsulates the MQSeries Message Queue Interface (MQI). There is a single C++ header file **imqi.hpp** which covers all of these classes.

For each class, the following information is shown:

Class hierarchy diagram
: A class diagram showing the class in its inheritance relation to its immediate parent classes, if any.

Other relevant classes
: These are document links to other relevant classes, such as parent classes, and the classes of objects used in method signatures.

Object attributes
: These are the attributes unique to the class, and are in addition to those attributes defined for any parent classes. Many attributes reflect MQSeries data structure members (see Appendix B, “MQI cross-reference” on page 95), and for detailed descriptions see the *MQSeries Application Programming Reference manual*.

Constructors
: These are the signatures of the special methods used to create an object of the class. See the glossary for further information.

Object methods (public)
: These are the signatures of methods that do require an instance of the class for their operation, and that have no usage restrictions.

Where it applies, the following information is also shown:

Class methods (public)
: These are the signatures of methods that do not require an instance of the class for their operation, and that have no usage restrictions.

Overloaded “(parent class)” methods
: These are the signatures of those virtual methods that are defined in parent classes, but exhibit different, polymorphic, behavior for this class.

Object methods (protected)
: These are the signatures of methods that do require an instance of the class for their operation, and are reserved for use by the implementations of derived classes. This section is of interest only to class writers, as opposed to class users.

Object data (protected)
: These are the implementation details for object instance data available to the implementations of derived classes. This section is of interest only to class writers, as opposed to class users.

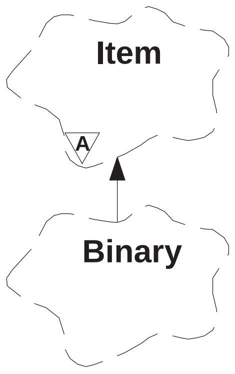
Reason codes
: These are the possible MQRC_* values (see Appendix C, “Reason codes” on page 105) that can be expected from those methods that can fail. For an exhaustive list of reason codes that can occur for an object of a given class, consult parent class documentation. The documented

list of reason codes for a given class does not include the reason codes for parent classes.

Notes

1. Objects of these classes are not thread-safe. This ensures optimal performance, but care must be taken not to access any given object from more than one thread.
2. For a multi-threaded program, use a separate ImqQueueManager object for each thread. MQSeries requires a separate queue manager connection for each thread, and does not permit cross-thread operations. Each ImqQueueManager object should have its own independent collection of ImqQueue and other objects, ensuring that objects in different threads are isolated from one another.

ImqBinary



This class encapsulates a binary byte array that can be used for ImqMessage **accounting token**, **correlation id**, and **message id** values. It allows easy assignment, copying, and comparison.

Other relevant classes

ImqItem (see "ImqItem" on page 43)
ImqMessage (see "ImqMessage" on page 45)

Object attributes

data An array of bytes of binary data. Initially null.

data length
The number of bytes. Initially zero.

data pointer
The address of the first byte of the **data**. Initially zero.

Constructors

ImqBinary( );
The default constructor.

ImqBinary(const ImqBinary & *binary* **);**
The copy constructor.

ImqBinary(const void * *data***, const size_t** *length* **);**
Copies *length* bytes from *data*.

Overloaded "ImqItem" methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Copies the **data** to the message buffer, replacing any existing content. Sets the *msg* **format** to MQFMT_NONE.

See the ImqItem class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Sets the **data** by transferring the remaining data from the message buffer, replacing the existing **data.**

To be successful, the ImqMessage **format** must be MQFMT_NONE.

See the ImqItem class method description for further details.

Object methods (public)

void operator = (const ImqBinary & *binary* **);**

Copies bytes from *binary*.

ImqBoolean operator == (const ImqBinary & *binary* **);**

Compares this object with *binary*. Returns zero if not equal and nonzero otherwise. The objects are equal if they have the same **data length** and the bytes match.

ImqBoolean copyOut(void * *buffer***, const size_t** *length***, const char** *pad* **= 0);**

Copies up to *length* bytes from the **data pointer** to *buffer*. If the **data length** is insufficient, the remaining space in *buffer* is filled with *pad* bytes. *buffer* may be zero if *length* is also zero. *length* must not be negative. Returns TRUE if successful.

size_t dataLength( ) const ;

Returns the **data length**.

ImqBoolean setDataLength(const size_t *length* **);**

Sets the **data length**. If the **data length** is changed as a result of this method, then the data in the object is uninitialized. Returns TRUE if successful.

void * dataPointer( ) const ;

Returns the **data pointer**.

ImqBoolean isNull( ) const ;

Returns TRUE if the **data length** is zero, or if all of the **data** bytes are zero. Otherwise returns FALSE.

ImqBoolean set(const void * *buffer***, const size_t** *length* **);**

Copies *length* bytes from *buffer*. Returns TRUE if successful.

Object methods (protected)

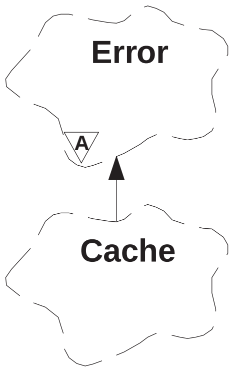
void clear( );

Reduces the **data length** to zero.

Reason codes

MQRC_NO_BUFFER
MQRC_STORAGE_NOT_AVAILABLE

ImqCache



Use this class to hold or marshal data in memory. The user can nominate a buffer of memory of fixed size, or the system can provide a flexible amount of memory automatically.

Other relevant classes

ImqError (see "ImqError" on page 33)

Object attributes

automatic buffer
Indicates whether buffer memory is managed automatically by the system (TRUE) or is supplied by the user (FALSE). Initially TRUE.

buffer length
The number of bytes of memory in the buffer. Initially zero.

buffer pointer
The address of the buffer memory. Initially null.

data length
The number of bytes succeeding the **data pointer**. Equal to or less than the **message length**. Initially zero.

data offset
The number of bytes preceding the **data pointer**. Equal to or less than the **message length**. Initially zero.

data pointer
The address of that part of the buffer that is to be written to or read from next. Initially null.

message length
The number of bytes of significant data in the buffer. Initially zero.

Constructors

ImqCache( );
The default constructor.

ImqCache(const ImqCache & *cache* **);**
The copy constructor.

Object methods (public)

void operator = (const ImqCache & *cache* **);**
Copies up to **message length** bytes of data from the *cache* object to the object. If **automatic buffer** is FALSE, then the **buffer length** must already be sufficient to accommodate the copied data.

ImqBoolean automaticBuffer( ) const ;
Returns the **automatic buffer** value.

size_t bufferLength( ) const ;
Returns the **buffer length**.

char * bufferPointer( ) const ;
Returns the **buffer pointer**.

void clearMessage( );
Sets the **message length** and **data offset** both to zero.

size_t dataLength( ) const ;
Returns the **data length**.

size_t dataOffset( ) const ;
Returns the **data offset**.

ImqBoolean setDataOffset(const size_t *offset* **);**
Sets the **data offset**. The **message length** is increased if necessary to ensure that it is no less than the **data offset**. Returns TRUE if successful.

char * dataPointer( ) const ;
Returns a copy of the **data pointer**.

size_t messageLength( ) const ;
Returns the **message length**.

ImqBoolean setMessageLength(const size_t *length* **);**
Sets the **message length**. Increases the **buffer length** if necessary to ensure that the **message length** is no greater than the **buffer length**. Reduces the **data offset** if necessary to ensure that it is no greater than the **message length**. Returns TRUE if successful.

ImqBoolean moreBytes(const size_t *bytes-required* **);**
Assures that *bytes-required* more bytes are available (for writing) between the **data pointer** and the end of the buffer. Returns TRUE if successful.

If **automatic buffer** is TRUE, then more memory will be acquired as required; otherwise, the **buffer length** must already be adequate.

ImqBoolean read(const size_t *length***, char * &** *external-buffer* **);**
Copies *length* bytes, from the buffer starting at the **data pointer** position, into the *external-buffer*. After the data has been copied, the **data offset** is increased by *length*. Returns TRUE if successful.

ImqBoolean resizeBuffer(const size_t *length* **);**
Varies the **buffer length**, provided that **automatic buffer** is TRUE. This is achieved by reallocating the buffer memory. Up to **message length** bytes of data from the existing buffer are copied to the new one. The maximum number copied is *length* bytes. The **buffer pointer** is changed. The **message length** and **data offset** are preserved as

closely as possible within the confines of the new buffer. Returns TRUE if successful. Returns FALSE if **automatic buffer** is FALSE.

Note: This method may fail with MQRC_STORAGE_NOT_AVAILABLE if there is any problem with system resources.

ImqBoolean useEmptyBuffer(const char * *external-buffer*, **const size_t** *length* **);**
Identifies an empty user buffer, setting the **buffer pointer** to point to *external-buffer*, the **buffer length** to *length*, and the **message length** to zero. Performs a **clearMessage**. If the buffer is fully primed with data, use the **useFullBuffer** method instead. If the buffer is partially primed with data, use the **setMessageLength** method to indicate the correct amount. Returns TRUE if successful.

This method can be used to identify a fixed amount of memory, as described above (*external-buffer* is non-null and *length* is non-zero), in which case **automatic buffer** is set to FALSE, or it can be used to revert to system-managed flexible memory (*external-buffer* is null and *length* is zero), in which case **automatic buffer** is set to TRUE.

ImqBoolean useFullBuffer(const char * *externalBuffer*, **const size_t** *length* **);**
As for **useEmptyBuffer**, except that the **message length** is set to *length*. Returns TRUE if successful.

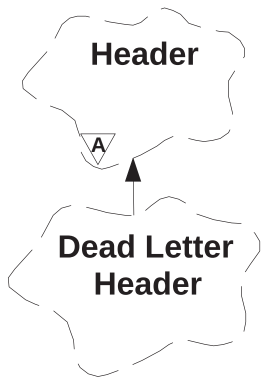
ImqBoolean write(const size_t *length*, **const char *** *external-buffer* **);**
Copies *length* bytes, from the *external-buffer*, into the buffer starting at the **data pointer** position. After the data has been copied, the **data offset** is increased by *length*, and the **message length** is increased if necessary to ensure that it is no less than the new **data offset** value. Returns TRUE if successful.

If **automatic buffer** is TRUE, an adequate amount of memory is guaranteed; otherwise, the ultimate **data offset** must not exceed the **buffer length**.

Reason codes

MQRC_BUFFER_NOT_AUTOMATIC
MQRC_DATA_TRUNCATED
MQRC_INSUFFICIENT_BUFFER
MQRC_INSUFFICIENT_DATA
MQRC_NULL_POINTER
MQRC_STORAGE_NOT_AVAILABLE
MQRC_ZERO_LENGTH

ImqDeadLetterHeader



This class encapsulates specific features of the MQDLH data structure (see Appendix B, “MQI cross-reference” on page 95). Objects of this class are typically used by an application that encounters an unprocessable message. A new message comprising a dead-letter header and the unprocessable message content is placed on the dead-letter queue, and the unprocessable message is discarded.

Other relevant classes

ImqHeader (see “ImqHeader” on page 38)
ImqItem (see “ImqItem” on page 43)
ImqMessage (see “ImqMessage” on page 45)
ImqString (see “ImqString” on page 82)

Object attributes

dead-letter reason code
The reason the message arrived on the dead-letter queue. Initially MQRC_NONE.

destination queue manager name
The name of the original destination queue manager. Initially null.

destination queue name
The name of the original destination queue. Initially null.

put application name
The name of the application that put the message on the dead-letter queue. Initially null.

put application type
The type of application that put the message on the dead-letter queue. Initially zero.

put date The date when the message was put on the dead-letter queue. Initially a null string.

put time The time when the message was put on the dead-letter queue. Initially a null string.

Constructors

ImqDeadLetterHeader( );
The default constructor.

ImqDeadLetterHeader(const ImqDeadLetterHeader & *header* **);**
The copy constructor.

Overloaded “ImqItem” methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Inserts an MQDLH data structure into the message buffer at the beginning, moving existing message data further along. Sets the *msg* **format** to MQFMT_DEAD_LETTER_HEADER.

See the ImqHeader class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Reads an MQDLH data structure from the message buffer.

To be successful, the ImqMessage **format** must be MQFMT_DEAD_LETTER_HEADER.

See the ImqHeader class method description for further details.

Object methods (public)

void operator = (const ImqDeadLetterHeader & *header* **);**
Instance data is copied from *header*, replacing the existing instance data.

MQLONG deadLetterReasonCode( ) const ;
Returns the **dead-letter reason code**.

void setDeadLetterReasonCode(const MQLONG *reason* **);**
Sets the **dead-letter reason code**.

ImqString destinationQueueManagerName( ) const ;
Returns the **destination queue manager name**.

void setDestinationQueueManagerName(const char * *name* **);**
Sets the **destination queue manager name**.

ImqString destinationQueueName( ) const ;
Returns a copy of the **destination queue name**.

void setDestinationQueueName(const char * *name* **);**
Sets the **destination queue name**.

ImqString putApplicationName( ) const ;
Returns a copy of the **put application name**.

void setPutApplicationName(const char * *name* **= 0);**
Sets the **put application name**.

MQLONG putApplicationType( ) const ;
Returns the **put application type**.

void setPutApplicationType(const MQLONG *type* **= MQAT_NO_CONTEXT);**
Sets the **put application type**.

ImqString putDate( ) const ;
Returns a copy of the **put date**.

void setPutDate(const char * *date* **= 0);**
Sets the **put date**.

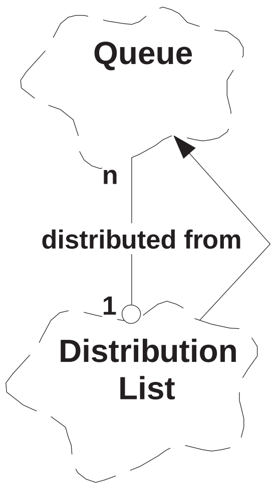
ImqString putTime( ) const ;
Returns a copy of the **put time**.

void setPutTime(const char * *time* **= 0);**
Sets the **put time**.

Object data (protected)

MQDLH *omqdlh*
The MQDLH data structure.

ImqDistributionList



This class encapsulates a distribution list.

Other relevant classes

ImqMessage (see “ImqMessage” on page 45)
ImqQueue (see “ImqQueue” on page 62)

Object attributes

first distributed queue
The first of one or more objects of class ImqQueue, in no particular order, in which the ImqQueue **distribution list reference** addresses this object. Initially zero.

Note: When an ImqDistributionList object is opened, any open ImqQueue objects that reference it are automatically closed.

Constructors

ImqDistributionList( );
The default constructor.

ImqDistributionList(const ImqDistributionList & *list* **);**
The copy constructor.

Object methods (public)

void operator = (const ImqDistributionList & *list* **);**
All ImqQueue objects that reference **this** object are dereferenced prior to copying. No ImqQueue objects will reference **this** object after the invocation of this method.

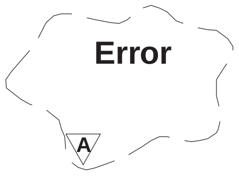
ImqQueue * firstDistributedQueue( ) const ;
Returns the **first distributed queue**.

Object methods (protected)

void setFirstDistributedQueue(ImqQueue * *queue* **= 0);**

Sets the **first distributed queue**.

ImqError



This abstract class provides information on errors associated with an object.

Other relevant classes

None.

Object attributes

completion code
The most recent completion code. Initially zero.

reason code
The most recent reason code. Initially zero.

Constructors

ImqError( );
The default constructor.

ImqError(const ImqError & *error* **);**
The copy constructor.

Object methods (public)

void operator = (const ImqError & *error* **);**
Instance data is copied from *error*, replacing the existing instance data.

void clearErrorCodes( );
Sets the **completion code** and **reason code** both to zero.

MQLONG completionCode( ) const ;
Returns the **completion code**.

MQLONG reasonCode( ) const ;
Returns the **reason code**.

Object methods (protected)

ImqBoolean checkReadPointer(const void * *pointer***, const size_t** *length* **);**
Verifies that the combination of pointer and length is valid for read-only access. Returns TRUE if successful.

ImqBoolean checkWritePointer(const void * *pointer***, const size_t** *length* **);**
Verifies that the combination of pointer and length is valid for read-write access. Returns TRUE if successful.

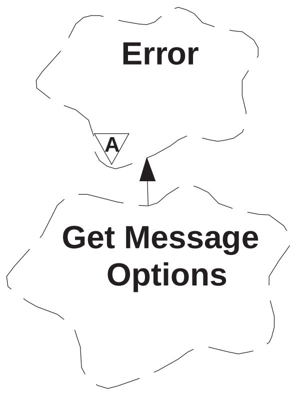
void setCompletionCode(const MQLONG *code* **= 0);**
Sets the **completion code**.

void setReasonCode(const MQLONG *code* **= 0);**
Sets the **reason code**.

Reason codes

MQRC_BUFFER_ERROR

ImqGetMessageOptions



This class encapsulates the MQGMO data structure (see Appendix B, "MQI cross-reference" on page 95).

Other relevant classes

ImqString (see "ImqString" on page 82)

Object attributes

group status
This is status of a message with respect to a group of messages. The initial value is MQGS_NOT_IN_GROUP.

match options
These are the options for selecting incoming messages. The initial value is MQMO_MATCH_MSG_ID or MQMO_MATCH_CORREL_ID.

options These are the options applicable to a message. The initial value is MQGMO_NO_WAIT.

resolved queue name
This attribute is read-only. This is the resolved queue name. Names are never longer than 48 characters and may be padded to that length with nulls. The initial value is a null string.

segmentation
The capability for segmentation of a message. The initial value is MSEG_INHIBITED.

segment status
The segmentation status of a message. The initial value is MQSS_NOT_A_SEGMENT.

sync-point participation
TRUE when messages are retrieved under sync-point control.

wait interval
This is the length of time that the ImqQueue class **get** method pauses while waiting for a suitable message to arrive, if one is not already available. The initial value is zero, which effects an indefinite wait. This attribute is ignored unless the **options** include MQGMO_WAIT.

Constructors

ImqGetMessageOptions( );
The default constructor.

ImqGetMessageOptions(const ImqGetMessageOptions & *gmo* **);**
The copy constructor.

Object methods (public)

void operator = (const ImqGetMessageOptions & *gmo* **);**
Instance data is copied from *gmo*, replacing the existing instance data.

MQCHAR groupStatus( ) const ;
Returns the **group status**.

void setGroupStatus(const MQCHAR *status* **);**
Sets the **group status**.

MQLONG matchOptions( ) const ;
Returns the **match options**.

void setMatchOptions(const MQLONG *options* **);**
Sets the **match options**.

MQLONG options( ) const ;
Returns the **options**.

void setOptions(const MQLONG *options* **);**
Sets the **options**, including the **sync-point participation** value.

ImqString resolvedQueueName( ) const ;
Returns a copy of the **resolved queue name**.

MQCHAR segmentation( ) const ;
Returns the **segmentation**.

void setSegmentation(const MQCHAR *value* **);**
Sets the **segmentation**.

MQCHAR segmentStatus( ) const ;
Returns the **segment status**.

void setSegmentStatus(const MQCHAR *status* **);**
Sets the **segment status**.

ImqBoolean syncPointParticipation( ) const ;
Returns the **sync-point participation** value, which is TRUE if the
| **options** include either MQGMO_SYNCPOINT or
| MQGMO_SYNCPOINT_IF_PERSISTENT.

void setSyncPointParticipation(const ImqBoolean *sync* **);**
| Sets the **sync-point participation** value. If *sync* is TRUE, the **options**
| are altered to include MQGMO_SYNCPOINT, and to exclude both
| MQGMO_NO_SYNCPOINT and
| MQGMO_SYNCPOINT_IF_PERSISTENT. If *sync* is FALSE, the
| **options** are altered to include MQGMO_NO_SYNCPOINT, and to
| exclude both MQGMO_SYNCPOINT and
| MQGMO_SYNCPOINT_IF_PERSISTENT.

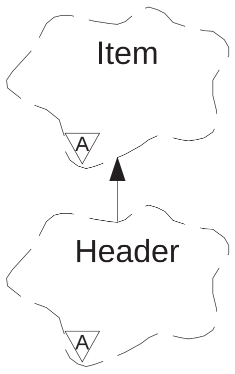
MQLONG waitInterval( ) const ;
Returns the **wait interval**.

void setWaitInterval(const MQLONG *interval* **);**
Sets the **wait interval**.

Object data (protected)

MQGMO *omqgmo*
The MQGMO data structure.

ImqHeader



This abstract class encapsulates common features of the MQDLH data structure (see Appendix B, “MQI cross-reference” on page 95).

Other relevant classes

ImqDeadLetterHeader (see “ImqDeadLetterHeader” on page 28)
ImqImsBridgeHeader (see “ImqImsBridgeHeader” on page 40)
ImqItem (see “ImqItem” on page 43)
ImqMessage (see “ImqMessage” on page 45)
ImqReferenceHeader (see “ImqReferenceHeader” on page 79)
ImqString (see “ImqString” on page 82)

Object attributes

character set
The original coded character set identifier. Initially MQCCSI_Q_MGR.

encoding The original encoding. Initially MQENC_NATIVE.

format The original format. Initially MQFMT_NONE.

header flags
The initial value is zero for objects of the ImqDeadLetterHeader class, MQIIH_NONE for objects of the ImqImsBridgeHeader class, and MQRMHF_LAST for objects of the ImqReferenceHeader class.

Constructors

ImqHeader( );
The default constructor.

ImqHeader(const ImqHeader & *header* **);**
The copy constructor.

Overloaded “ImqItem” methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **) = 0 ;**
Before the data structure is written, the **encoding**, **character set**, and **format** from the *msg* object are copied into this object, and the **encoding** and **character set** of the *msg* object are set to MQENC_NATIVE and MQCCSI_Q_MGR respectively. Thus the header attributes reflect the message data that will follow after the data written to the message buffer.

See the ImqItem class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **) = 0 ;**
After a successful data transfer, the **encoding**, **character set** and **format** attributes from the header in the message buffer are copied to the *msg* object so that remaining items of data in the message buffer are correctly represented by the *msg* object attributes.

See the ImqItem class method description for further details.

Object methods (public)

void operator = (const ImqHeader & *header* **);**
Instance data is copied from *header*, replacing the existing instance data.

virtual MQLONG characterSet( ) const ;
Returns the **character set**.

virtual void setCharacterSet(const MQLONG *ccsid* **= MQCCSI_Q_MGR);**
Sets the **character set**.

virtual MQLONG encoding( ) const ;
Returns the **encoding**.

virtual void setEncoding(const MQLONG *encoding* **= MQENC_NATIVE);**
Sets the **encoding**.

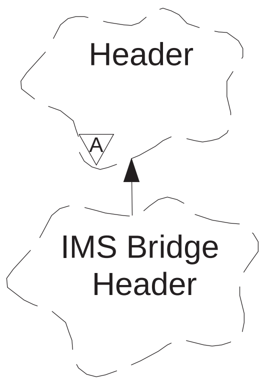
virtual ImqString format( ) const ;
Returns a copy of the **format**, including trailing blanks.

virtual void setFormat(const char * *name* **= 0);**
Sets the **format**, padding to 8 characters with trailing blanks.

virtual MQLONG headerFlags( ) const ;
Returns the **header flags**.

virtual void setHeaderFlags(const MQLONG *flags* **= 0);**
Sets the **header flags**.

ImqImsBridgeHeader



This class encapsulates specific features of the MQIIH data structure. Objects of this class are used by applications that send messages to the IMS bridge through MQSeries for MVS/ESA.

Note: The ImqHeader **character set** and **encoding** must have default values and must not be set to any other values.

Other relevant classes

ImqBinary (see "ImqBinary" on page 23)
ImqHeader (see "ImqHeader" on page 38)
ImqItem (see "ImqItem" on page 43)
ImqMessage (see "ImqMessage" on page 45)
ImqString (see "ImqString" on page 82)

Object attributes

authenticator
This is the RACF password or passticket, of length MQ_AUTHENTICATOR_LENGTH. The initial value is MQIAUT_NONE.

commit mode
This is the commit mode. See the *OTMA User's Guide* for more information about IMS commit modes. The initial value is MQICM_COMMIT_THEN_SEND.

logical terminal override
This is the logical terminal override, of length MQ_LTERM_OVERRIDE_LENGTH. The initial value is a null string.

message format services map name
This is the MFS map name, of length MQ_MFS_MAP_NAME_LENGTH. The initial value is a null string.

reply-to format
This is the format of any reply, of length MQ_FORMAT_LENGTH. The initial value is MQFMT_NONE.

security scope
This indicates the desired IMS security processing. The initial value is MQISS_CHECK.

transaction instance id
This is the transaction instance identity, a binary (MQBYTE16) value of length MQ_TRAN_INSTANCE_ID_LENGTH. The initial value is MQITII_NONE.

transaction state
This indicates the state of the IMS conversation. The initial value is MQITS_NOT_IN_CONVERSATION.

Constructors

ImqImsBridgeHeader( );
The default constructor.

ImqImsBridgeHeader(const ImqImsBridgeHeader & *header* **);**
The copy constructor.

Overloaded "ImqItem" methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Inserts an MQIIH data structure into the message buffer at the beginning, moving existing message data further along. Sets the *msg* **format** to MQFMT_IMS.

See the parent class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Reads an MQIIH data structure from the message buffer.

To be successful, the **encoding** of the *msg* object should be MQENC_NATIVE. It is recommended that messages be retrieved with MQGMO_CONVERT to MQENC_NATIVE.

To be successful, the ImqMessage **format** must be MQFMT_IMS.

See the parent class method description for further details.

Object methods (public)

void operator = (const ImqImsBridgeHeader & *header* **);**
Instance data is copied from *header*, replacing the existing instance data.

ImqString authenticator( ) const ;
Returns a copy of the **authenticator**, padded with trailing blanks to length MQ_AUTHENTICATOR_LENGTH.

void setAuthenticator(const char * *name* **);**
Sets the **authenticator**.

MQCHAR commitMode( ) const ;
Returns the **commit mode**.

void setCommitMode(const MQCHAR *mode* **);**
Sets the **commit mode**.

ImqString logicalTerminalOverride( ) const ;
Returns a copy of the **logical terminal override**.

void setLogicalTerminalOverride(const char * *override* **);**
Sets the **logical terminal override**.

ImqString messageFormatServicesMapName( ) const ;
Returns a copy of the **message format services map name**.

void setMessageFormatServicesMapName(const char * *name* **);**
Sets the **message format services map name**.

ImqString replyToFormat( ) const ;
Returns a copy of the **reply-to format**, padded with trailing blanks to length MQ_FORMAT_LENGTH.

void setReplyToFormat(const char * *format* **);**
Sets the **reply-to format**, padding with trailing blanks to length MQ_FORMAT_LENGTH.

MQCHAR securityScope( ) const ;
Returns the **security scope**.

void setSecurityScope(const MQCHAR *scope* **);**
Sets the **security scope**.

ImqBinary transactionInstanceId( ) const ;
Returns a copy of the **transaction instance id**.

ImqBoolean setTransactionInstanceId(const ImqBinary & *id* **);**
Sets the **transaction instance id**. The **data length** of *token* must be either zero or MQ_TRAN_INSTANCE_ID_LENGTH. Returns TRUE if successful.

void setTransactionInstanceId(const MQBYTE16 *id* **= 0);**
Sets the **transaction instance id**. *id* may be zero, which is the same as specifying MQITII_NONE. If *id* is non-zero, then it must address MQ_TRAN_INSTANCE_ID_LENGTH bytes of binary data. When using pre-defined values such as MQITII_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQITII_NONE.

MQCHAR transactionState( ) const ;
Returns the **transaction state**.

void setTransactionState(const MQCHAR *state* **);**
Sets the **transaction state**.

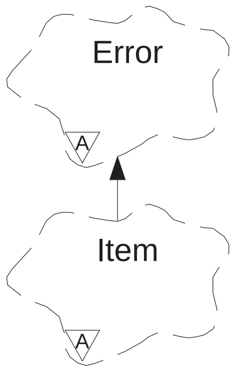
Object data (protected)

MQIIH *omqiih*
The MQIIH data structure.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR

ImqItem



This abstract class represents an item, perhaps one of several, within a message. Items are concatenated together in a message buffer. Each specialization is associated with a particular data structure that begins with a structure id.

Polymorphic methods in this abstract class allow items to be copied to and from messages. The ImqMessage class **readItem** and **writeItem** methods provide another style of invoking these polymorphic methods, a style that is more natural for application programs.

Other relevant classes

ImqCache (see "ImqCache" on page 25)
ImqError (see "ImqError" on page 33)
ImqMessage (see "ImqMessage" on page 45)

Object attributes

structure id
This attribute is read-only. A string of 4 characters at the beginning of the data structure.

Constructors

ImqItem( );
The default constructor.

ImqItem(const ImqItem & *item* **);**
The copy constructor.

Class methods (public)

static ImqBoolean structureIdIs(const char * *structure-id-to-test*, **const ImqMessage &** *msg* **);**
Returns TRUE if the **structure id** of the next ImqItem in the incoming *msg* is the same as *structure-id-to-test*. The next item is identified as that part of the message buffer currently addressed by the ImqCache **data pointer**.

Object methods (public)

void operator = (const ImqItem & ***item*** **);**

Instance data is copied from *item*, replacing the existing instance data.

virtual ImqBoolean copyOut(ImqMessage & ***msg*** **) = 0 ;**

Writes this object as the next item in an outgoing message buffer, appending it to any existing items. If the write operation is successful, the ImqCache **data length** is increased. Returns TRUE if successful.

Override this method to work with a specific subclass.

virtual ImqBoolean pasteIn(ImqMessage & ***msg*** **) = 0 ;**

Reads this object destructively[2] from the incoming message buffer.

The (sub)class of this object must be consistent with the **structure id** found next in the message buffer of the *msg* object.

The **encoding** of the *msg* object should be MQENC_NATIVE. It is recommended that messages be retrieved with the ImqMessage **encoding** set to MQENC_NATIVE, and with the ImqGetMessageOptions **options**.

If the read operation is successful, the ImqCache **data length** is reduced. Returns TRUE if successful.

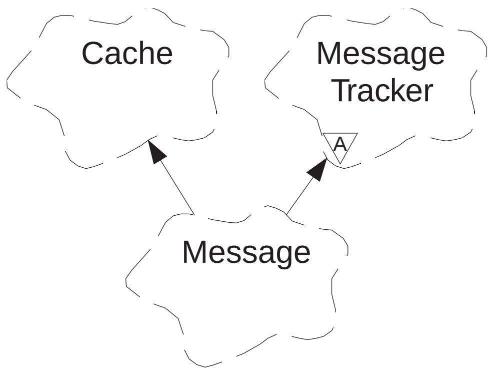
Override this method to work with a specific subclass.

Reason codes

MQRC_ENCODING_ERROR
MQRC_STRUC_ID_ERROR
MQRC_INCONSISTENT_FORMAT
MQRC_INSUFFICIENT_BUFFER
MQRC_INSUFFICIENT_DATA

[2] The read is destructive in that the ImqCache **data pointer** is moved on. However, the buffer content remains the same, so data can be re-read by resetting the ImqCache **data pointer**.

ImqMessage



This class encapsulates an MQMD data structure (see Appendix B, “MQI cross-reference” on page 95), and also handles the construction and reconstruction of message data.

Other relevant classes

ImqCache (see “ImqCache” on page 25)
ImqItem (see “ImqItem” on page 43)
ImqMessageTracker (see “ImqMessageTracker” on page 50)
ImqString (see “ImqString” on page 82)

Object attributes

application id data
Identity information associated with a message. The initial value is a null string.

application origin data
Origin information associated with a message. The initial value is a null string.

backout count
This attribute is read-only. The number of times a message has been tentatively retrieved and subsequently backed out. The initial value is zero.

character set
Coded Character Set Id. The initial value is MQCCSI_Q_MGR.

encoding The machine encoding of the message data. The initial value is MQENC_NATIVE.

expiry A time-dependent quantity that controls how long MQSeries retains an unretrieved message before discarding it. The initial value is MQEI_UNLIMITED.

format The name of the format (template) that describes the layout of data in the buffer. Names longer than 8 characters are truncated to 8 characters. Names are always padded with blanks to 8 characters. The initial value is MQFMT_NONE.

message flags
Segmentation control information. The initial value is MQMF_SEGMENTATION_INHIBITED.

message type
The broad categorization of a message. The initial value is MQMT_DATAGRAM.

offset Offset information. The initial value is 0.

original length
The original length of a segmented message. The initial value is MQOL_UNDEFINED.

persistence
Indicates that the message is important and must at all times be backed up using persistent storage. This option implies a performance penalty. The initial value is MQPER_PERSISTENCE_AS_Q_DEF.

priority The relative priority for transmission and delivery. Messages of the same priority are usually delivered in the same sequence as they were supplied (although there are several criteria that must be satisfied to guarantee this). The initial value is MQPRI_PRIORITY_AS_Q_DEF.

put application name
The name of the application that put a message. Initially a null string.

put application type
The type of application that put a message. The initial value is MQAT_NO_CONTEXT.

put date The date on which a message was put. Initially a null string.

put time The time at which a message was put. Initially a null string.

reply-to queue manager name
The name of the queue manager to which any reply should be sent. Initially a null string.

reply-to queue name
The name of the queue to which any reply should be sent. Initially a null string.

report Feedback information associated with a message. The initial value is MQRO_NONE.

sequence number
Sequence information identifying a message within a group. The initial value is 1.

| **total message length**
| This attribute is read-only. This is the number of bytes that were
| available during the most recent attempt to read a message. This
| number will be greater than the ImqCache **message length** if the last
| message was truncated, or if the last message was not read because
| truncation would have occurred. The initial value is zero.

| This attribute can be useful in any situation involving truncated
| messages.

user id A user identity associated with a message. Initially a null string.

Constructors

ImqMessage( );
: The default constructor.

ImqMessage(const ImqMessage & *msg* **);**
: The copy constructor. See the **operator =** method for details.

Object methods (public)

void operator = (const ImqMessage & *msg* **);**
: Copies the MQMD and message data from *msg*. If a buffer has been supplied by the user for this object, the amount of data copied is restricted to the available buffer size. Otherwise, the system ensures that a buffer of adequate size is made available for the copied data.

ImqString applicationIdData( ) const ;
: Returns a copy of the **application id data**.

void setApplicationIdData(const char * *data* **= 0);**
: Sets the **application id data**.

ImqString applicationOriginData( ) const ;
: Returns a copy of the **application origin data**.

void setApplicationOriginData(const char * *data* **= 0);**
: Sets the **application origin data**.

MQLONG backoutCount( ) const ;
: Returns the **backout count**.

MQLONG characterSet( ) const ;
: Returns the **character set**.

void setCharacterSet(const MQLONG *ccsid* **= MQCCSI_Q_MGR);**
: Sets the **character set**.

MQLONG encoding( ) const ;
: Returns the **encoding**.

void setEncoding(const MQLONG *encoding* **= MQENC_NATIVE);**
: Sets the **encoding**.

MQLONG expiry( ) const ;
: Returns the **expiry**.

void setExpiry(const MQLONG *expiry* **);**
: Sets the **expiry**.

ImqString format( ) const ;
: Returns a copy of the **format**, including trailing blanks.

ImqBoolean formatIs(const char * *format-to-test* **) const ;**
: Returns TRUE if the **format** is the same as *format-to-test*.

void setFormat(const char * *name* **= 0);**
: Sets the **format**, padding to 8 characters with trailing blanks.

MQLONG messageFlags( ) const ;
: Returns the **message flags**.

void setMessageFlags(const MQLONG *flags* **);**
: Sets the **message flags**.

MQLONG messageType( ) const ;
Returns the **message type**.

void setMessageType(const MQLONG *type* **);**
Sets the **message type**.

MQLONG offset( ) const ;
Returns the **offset**.

void setOffset(const MQLONG *offset* **);**
Sets the **offset**.

MQLONG originalLength( ) const ;
Returns the **original length**.

void setOriginalLength(const MQLONG *length* **);**
Sets the **original length**.

MQLONG persistence( ) const ;
Returns the **persistence**.

void setPersistence(const MQLONG *persistence* **);**
Sets the **persistence**.

MQLONG priority( ) const ;
Returns the **priority**.

void setPriority(const MQLONG *priority* **);**
Sets the **priority**.

ImqString putApplicationName( ) const ;
Returns a copy of the **put application name**.

void setPutApplicationName(const char * *name* **= 0);**
Sets the **put application name**.

MQLONG putApplicationType( ) const ;
Returns the **put application type**.

void setPutApplicationType(const MQLONG *type* **= MQAT_NO_CONTEXT);**
Sets the **put application type**.

ImqString putDate( ) const ;
Returns a copy of the **put date**.

void setPutDate(const char * *date* **= 0);**
Sets the **put date**.

ImqString putTime( ) const ;
Returns a copy of the **put time**.

void setPutTime(const char * *time* **= 0);**
Sets the **put time**.

ImqBoolean readItem(ImqItem & *item* **);**
Reads into the *item* object from the message buffer, using the ImqItem **pasteIn** method. Returns TRUE if successful.

ImqString replyToQueueManagerName( ) const ;
Returns a copy of the **reply-to queue manager name**.

void setReplyToQueueManagerName(const char * *name* **= 0);**
Sets the **reply-to queue manager name**.

ImqString replyToQueueName( ) const ;
Returns a copy of the **reply-to queue name**.

void setReplyToQueueName(const char * *name* **= 0);**
Sets the **reply-to queue name**.

MQLONG report( ) const ;
Returns the **report**.

void setReport(const MQLONG *report* **);**
Sets the **report**.

MQLONG sequenceNumber( ) const ;
Returns the **sequence number**.

void setSequenceNumber(const MQLONG *number* **);**
Sets the **sequence number**.

| **size_t totalMessageLength( ) const ;**
| Returns the **total message length**.

ImqString userId( ) const ;
Returns a copy of the **user id**.

void setUserId(const char * *id* **= 0);**
Sets the **user id**.

ImqBoolean writeItem(ImqItem & *item* **);**
Writes from the *item* object into the message buffer, using the ImqItem **copyOut** method. Writing may take the form of insertion, replacement or an append: this depends on the class of the *item* object. Returns TRUE if successful.

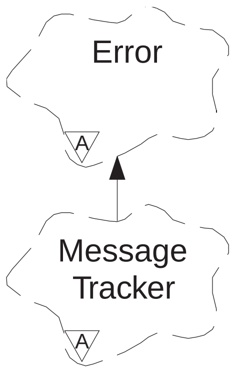
Object data (protected)

MQMD *omqmd*
The MQMD data structure.

Reason codes

MQRC_ENCODING_ERROR
MQRC_STRUC_ID_ERROR
MQRC_INCONSISTENT_FORMAT
MQRC_INSUFFICIENT_BUFFER
MQRC_INSUFFICIENT_DATA

ImqMessageTracker



This abstract class encapsulates those attributes of an ImqMessage or ImqQueue object that can be associated with either object.

Other relevant classes

ImqBinary (see "ImqBinary" on page 23)
ImqError (see "ImqError" on page 33)
ImqMessage (see "ImqMessage" on page 45)
ImqQueue (see "ImqQueue" on page 62)

Object attributes

accounting token
A binary value (MQBYTE32) of length MQ_ACCOUNTING_TOKEN_LENGTH. The initial value is MQACT_NONE.

correlation id
A binary value (MQBYTE24) of length MQ_CORREL_ID_LENGTH assigned by the user for the purpose of correlating messages. The initial value is MQCI_NONE.

feedback Feedback information to be sent with a message. The initial value is MQFB_NONE.

group id A binary value (MQBYTE24) of length MQ_GROUP_ID_LENGTH unique within a queue. The initial value is MQGI_NONE.

message id
A binary value (MQBYTE24) of length MQ_MSG_ID_LENGTH unique within a queue. The initial value is MQMI_NONE.

Constructors

ImqMessageTracker( );
The default constructor.

ImqMessageTracker(const ImqMessageTracker & *tracker* **);**
The copy constructor. See the **operator =** method for details.

Object methods (public)

void operator = (const ImqMessageTracker & *tracker* **);**
Instance data is copied from *tracker*, replacing the existing instance data.

ImqBinary accountingToken( ) const ;
Returns a copy of the **accounting token**.

ImqBoolean setAccountingToken(const ImqBinary & *token* **);**
Sets the **accounting token**. The **data length** of *token* must be either zero or MQ_ACCOUNTING_TOKEN_LENGTH. Returns TRUE if successful.

void setAccountingToken(const MQBYTE32 *token* **= 0);**
Sets the **accounting token**. *token* may be zero, which is the same as specifying MQACT_NONE. If *token* is non-zero, then it must address MQ_ACCOUNTING_TOKEN_LENGTH bytes of binary data. When using predefined values such as MQACT_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQACT_NONE.

ImqBinary correlationId( ) const ;
Returns a copy of the **correlation id**.

ImqBoolean setCorrelationId(const ImqBinary & *token* **);**
Sets the **correlation id**. The **data length** of *token* must be either zero or MQ_CORREL_ID_LENGTH. Returns TRUE if successful.

void setCorrelationId(const MQBYTE24 *id* **= 0);**
Sets the **correlation id**. *id* may be zero, which is the same as specifying MQCI_NONE. If *id* is non-zero, then it must address MQ_CORREL_ID_LENGTH bytes of binary data. When using pre-defined values such as MQCI_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQCI_NONE.

MQLONG feedback( ) const ;
Returns the **feedback**.

void setFeedback(const MQLONG *feedback* **);**
Sets the **feedback**.

ImqBinary groupId( ) const ;
Returns a copy of the **group id**.

ImqBoolean setGroupId(const ImqBinary & *token* **);**
Sets the **group id**. The **data length** of *token* must be either zero or MQ_GROUP_ID_LENGTH. Returns TRUE if successful.

void setGroupId(const MQBYTE24 *id* **= 0);**
Sets the **group id**. *id* may be zero, which is the same as specifying MQGI_NONE. If *id* is non-zero, it must address MQ_GROUP_ID_LENGTH bytes of binary data. When using pre-defined values such as MQGI_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQGI_NONE.

ImqBinary messageId( ) const ;
Returns a copy of the **message id**.

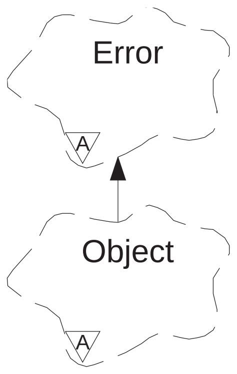
ImqBoolean setMessageId(const ImqBinary & *token* **);**
Sets the **message id**. The **data length** of *token* must be either zero or MQ_MSG_ID_LENGTH. Returns TRUE if successful.

void setMessageId(const MQBYTE24 *id* **= 0);**
Sets the **message id**. *id* may be zero, which is the same as specifying MQMI_NONE. If *id* is non-zero, it must address MQ_MSG_ID_LENGTH bytes of binary data. When using pre-defined values such as MQMI_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQMI_NONE.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR

ImqObject



This class is abstract. When an object of this class is destroyed, it is automatically closed, and its ImqQueueManager connection severed.

Other relevant classes

ImqError (see "ImqError" on page 33)
ImqQueueManager (see "ImqQueueManager" on page 73)
ImqString (see "ImqString" on page 82)

Object attributes

alternate user id
Up to MQ_USER_ID_LENGTH characters. The initial value is a null string.

close options
The initial value is MQCO_NONE. This attribute is ignored during implicit reopen operations, where a value of MQCO_NONE is always used.

connection reference
A reference to an ImqQueueManager object that provides the required connection to a (local) queue manager. For an ImqQueueManager object, it will be the object itself. Initially null.

Note: Do not confuse this with the ImqQueue **queue manager name** that identifies a queue manager (possibly remote) for a named queue.

description
This attribute is read-only. The descriptive name (up to 64 characters) of the queue manager, queue or process.

name The name (up to 48 characters) of the queue manager, queue or process, as appropriate. The initial value is a null string. The name of a model queue changes after an **open** to the name of the resulting dynamic queue. An actual queue manager name is always returned in place of a null queue manager name.

next managed object
This is the next object of this class, in no particular order, having the same **connection reference** as this object. Initially zero.

open options

The initial value is MQOO_INQUIRE. There are two ways to set appropriate values:

1. Do not set the **open options** and do not use the **open** method. MQSeries automatically adjusts the **open options** and automatically opens, reopen and closees objects as required. This may result in unnecessary reopen operations, because MQSeries uses the **openFor** method, and this adds **open options** incrementally only.
2. Set the **open options** as appropriate before using any methods that result in an MQI call (see Appendix B, "MQI cross-reference" on page 95). This ensures that unnecessary reopen operations do not occur. It is strongly recommended that the open options be set explicitly if any of the potential reopen (see "Reopen" on page 15) problems are likely to occur.

   If you use the **open** method, you ***must*** ensure that the **open options** are appropriate first. However, using the **open** method is not mandatory; MQSeries still exhibits the same behavior as in case 1, but in this circumstance the behavior is efficient.

Zero is not a valid value, and so the appropriate value must be set before attempting to open the object. This can be done either using **setOpenOptions**(*lOpenOptions*) followed by **open**( ), or by using **openFor**(*lRequiredOpenOption*).

open status

This attribute is read-only. Indicates whether the object is open (TRUE) or closed (FALSE). Initially FALSE.

previous managed object

This is the previous object of this class, in no particular order, having the same **connection reference** as this object. Initially zero.

Constructors

ImqObject( );

The default constructor.

ImqObject(const ImqObject & *object* **);**

The copy constructor. The **open status** will be FALSE.

Object methods (public)

void operator = (const ImqObject & *object* **);**

Performs a close if necessary, and then copies the instance data from *object*. The **open status** will be FALSE.

ImqString alternateUserId( ) const ;

Returns a copy of the **alternate user id**.

ImqBoolean setAlternateUserId(const char * *id* **);**

Sets the **alternate user id**. The **alternate user id** can only be set while the **open status** is FALSE. Returns TRUE if successful.

ImqBoolean close( );

Sets the **open status** to FALSE. Returns TRUE if successful.

MQLONG closeOptions( ) const ;

Returns the **close options**.

void setCloseOptions(const MQLONG *options* **);**
Sets the **close options**.

ImqQueueManager * connectionReference( ) const ;
Returns the **connection reference**.

void setConnectionReference(ImqQueueManager & *manager* **);**
Sets the **connection reference**.

void setConnectionReference(ImqQueueManager * *manager* **= 0);**
Sets the **connection reference**.

virtual ImqBoolean description(ImqString & *description* **) = 0 ;**
Provides a copy of the **description**. Returns TRUE if successful.

ImqString description( );
Returns a copy of the **description** without any indication of possible errors.

virtual ImqBoolean name(ImqString & *name* **);**
Provides a copy of the **name**. Returns TRUE if successful.

ImqString name( );
Returns a copy of the **name** without any indication of possible errors.

ImqBoolean setName(const char * *name* **= 0);**
Sets the **name**. The **name** can only be set while the **open status** is FALSE, and, for an ImqQueueManager, while the **connection status** is FALSE. Returns TRUE if successful.

| **ImqObject * nextManagedObject( ) const ;**
| Returns the **next managed object**.

ImqBoolean open( );
Changes the **open status** to TRUE by opening the object as necessary, using amongst other attributes the **open options** and the **name**. Uses the **connection reference** information and the ImqQueueManager **connect** method if necessary to ensure that the ImqQueueManager **connection status** is TRUE. Returns the **open status**.

ImqBoolean openFor(const MQLONG *required-options* **= 0);**
Attempts to ensure that the object is open with **open options** that include the *required-options* specified.

If *required-options* is zero, it is assumed that input is required, and that any input option will suffice. So, if the **open options** already contain one of:

MQOO_INPUT_AS_Q_DEF
MQOO_INPUT_SHARED
MQOO_INPUT_EXCLUSIVE

then the **open options** are already satisfactory and will not be changed; if the **open options** do not already contain any of the above, then MQOO_INPUT_AS_Q_DEF will be set in the **open options**.

If *required-options* is non-zero, then the required options are added to the **open options**; if *required-options* is any of the above, then the others are reset.

If any of the **open options** are changed and the object is already open, then the object will be closed temporarily and reopened in order to adjust the **open options**.

Returns TRUE if successful. Success indicates that the object is open with appropriate options.

MQLONG openOptions( ) const ;
Returns the **open options**.

ImqBoolean setOpenOptions(const MQLONG *options* **);**
Sets the **open options**. The **open options** can only be set while the **open status** is FALSE. Returns TRUE if successful.

ImqBoolean openStatus( ) const ;
Returns the **open status**.

ImqObject * previousManagedObject( ) const ;
Returns the **previous managed object**.

Object methods (protected)

virtual ImqBoolean closeTemporarily( );
Closes an object safely prior to reopening. Returns TRUE if successful.

Note: This method assumes that the **open status** is TRUE.

MQHCONN connectionHandle( ) const ;
Returns the MQHCONN associated with the **connection reference**. This value will be zero if there is no **connection reference** or if the ImqQueueManager is not connected.

ImqBoolean inquire(const MQLONG *int-attr***, MQLONG &** *value* **);**
Returns an integer value, the index of which is an MQIA_* value. In case of error, the value is set to MQIAV_UNDEFINED.

ImqBoolean inquire(const MQLONG *char-attr***, char * &** *buffer***, const size_t** *length* **);**
Returns a character string, the index of which is an MQCA_* value.

Note: Both of the above methods return only a single attribute value. If a “snapshot” is required of more than one value, where the values are consistent with each other for an instant, MQSeries C++ does not provide this facility and it is necessary to use the MQINQ call with appropriate parameters.

virtual void openInformationDisperse( );
Disperses information from the variable section of the MQOD data structure immediately after an MQOPEN call.

virtual ImqBoolean openInformationPrepare( );
Prepares information for the variable section of the MQOD data structure immediately prior to an MQOPEN call. Returns TRUE if successful.

ImqBoolean set(const MQLONG *int-attr***, const MQLONG** *value* **);**
Sets an MQSeries integer attribute.

ImqBoolean set(const MQLONG *char-attr***, const char *** *buffer***, const size_t** *required-length* **);**
Sets an MQSeries character attribute.

void setNextManagedObject(const ImqObject * *object* **= 0);**
Sets the **next managed object**.

void setPreviousManagedObject(const ImqObject * *object* **= 0);**
Sets the **previous managed object**.

Object data (protected)

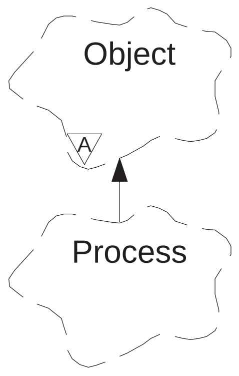
MQHOBJ *ohobj*
The MQSeries object handle (only valid when **open status** is TRUE).

MQOD *omqod*
The embedded MQOD data structure.

Reason codes

MQRC_ATTRIBUTE_LOCKED
MQRC_INCONSISTENT_OBJECT_STATE
MQRC_NO_CONNECTION_REFERENCE
MQRC_STORAGE_NOT_AVAILABLE
(reason codes from MQCLOSE)
(reason codes from MQCONN)
(reason codes from MQINQ)
(reason codes from MQOPEN)
(reason codes from MQSET)

ImqProcess



This class encapsulates an application process (an MQSeries object or type MQOT_PROCESS) that can be triggered by a trigger monitor.

Other relevant classes

ImqObject (see "ImqObject" on page 53)

Object attributes

application id
: This attribute is read-only. This is the identity of the application process.

application type
: This attribute is read-only. This is the type of the application process.

environment data
: This attribute is read-only. This is the environment information for the process.

user data
: This attribute is read-only. This is user data for the process.

Constructors

ImqProcess( );
: The default constructor.

ImqProcess(const ImqProcess & *process* **);**
: The copy constructor. The ImqObject **open status** will be FALSE.

ImqProcess(const char * *name* **);**
: Sets the ImqObject **name**.

Object methods (public)

void operator = (const ImqProcess & *process* **);**
: Performs a close if necessary, and then copies instance data from *process*. The ImqObject **open status** will be FALSE.

ImqBoolean applicationId(ImqString & *id* **);**
: Provides a copy of the **application id**. Returns TRUE if successful.

ImqString applicationId( );
: Returns the **application id** without any indication of possible errors.

ImqBoolean applicationType(MQLONG & *type* **);**
Provides a copy of the **application type**. Returns TRUE if successful.

MQLONG applicationType( );
Returns the **application type** without any indication of possible errors.

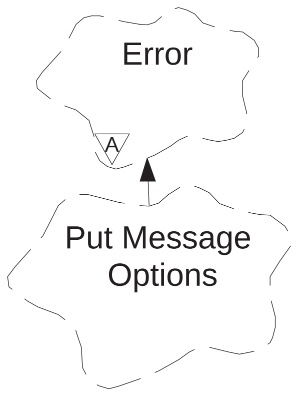
ImqBoolean environmentData(ImqString & *data* **);**
Provides a copy of the **environment data**. Returns TRUE if successful.

ImqString environmentData( );
Returns the **environment data** without any indication of possible errors.

ImqBoolean userData(ImqString & *data* **);**
Provides a copy of the **user data**. Returns TRUE if successful.

ImqString userData( );
Returns the **user data** without any indication of possible errors.

ImqPutMessageOptions



This class encapsulates the MQPMO data structure (see Appendix B, “MQI cross-reference” on page 95).

Other relevant classes

ImqError (see “ImqError” on page 33)
ImqMessage (see “ImqMessage” on page 45)
ImqQueue (see “ImqQueue” on page 62)
ImqString (see “ImqString” on page 82)

Object attributes

context reference
An ImqQueue that provides a context for messages. Initially there is no reference.

options These are the put message options. The initial value is MQPMO_NONE.

record fields
These are the flags that control the inclusion of put message records when a message is put. The initial value is MQPMRF_NONE.

ImqMessageTracker attributes are taken from the ImqQueue object for any field that is specified. ImqMessageTracker attributes are taken from the ImqMessage object for any field that is *not* specified.

resolved queue manager name
This attribute is read-only. This is the name of a destination queue manager determined during a put. Initially null.

resolved queue name
This attribute is read-only. This is the name of a destination queue determined during a put. Initially null.

sync-point participation
TRUE when messages are put under sync-point control.

Constructors

ImqPutMessageOptions( );
The default constructor.

ImqPutMessageOptions(const ImqPutMessageOptions & *pmo* **);**
The copy constructor.

Object methods (public)

void operator = (const ImqPutMessageOptions & *pmo* **);**
Instance data is copied from *pmo*, replacing the existing instance data.

ImqQueue * contextReference( ) const ;
Returns the **context reference**.

void setContextReference(const ImqQueue & *queue* **);**
Sets the **context reference**.

void setContextReference(const ImqQueue * *queue* **= 0);**
Sets the **context reference**.

MQLONG options( ) const ;
Returns the **options**.

void setOptions(const MQLONG *options* **);**
Sets the **options**, including the **sync-point participation** value.

MQLONG recordFields( ) const ;
Returns the **record fields**.

void setRecordFields(const MQLONG *fields* **);**
Sets the **record fields**.

ImqString resolvedQueueManagerName( ) const ;
Returns a copy of the **resolved queue manager name**.

ImqString resolvedQueueName( ) const ;
Returns a copy of the **resolved queue name**.

ImqBoolean syncPointParticipation( ) const ;
Returns the **sync-point participation** value, which is TRUE if the **options** include MQPMO_SYNCPOINT.

void setSyncPointParticipation(const ImqBoolean *sync* **);**
| Sets the **sync-point participation** value. If *sync* is TRUE, the **options**
| are altered to include MQPMO_SYNCPOINT, and to exclude
| MQPMO_NO_SYNCPOINT. If *sync* is FALSE, the **options** are altered
| to include MQPMO_NO_SYNCPOINT, and to exclude
| MQPMO_SYNCPOINT.

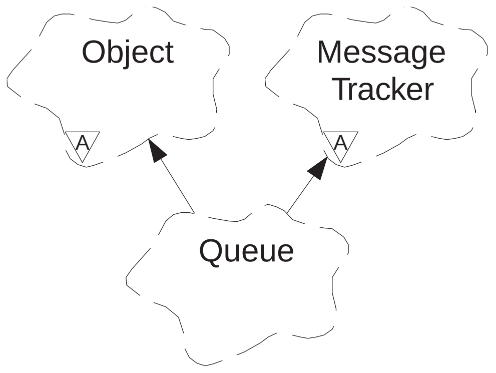
Object data (protected)

MQPMO *omqpmo*
The MQPMO data structure.

Reason codes

MQRC_STORAGE_NOT_AVAILABLE

ImqQueue



This class encapsulates a message queue (an MQSeries object or type MQOT_Q).

Other relevant classes

ImqCache (see "ImqCache" on page 25)
ImqDistributionList (see "ImqDistributionList" on page 31)
ImqGetMessageOptions (see "ImqGetMessageOptions" on page 35)
ImqMessage (see "ImqMessage" on page 45)
ImqMessageTracker (see "ImqMessageTracker" on page 50)
ImqObject (see "ImqObject" on page 53)
ImqPutMessageOptions (see "ImqPutMessageOptions" on page 60)
ImqQueueManager (see "ImqQueueManager" on page 73)
ImqString (see "ImqString" on page 82)

Object attributes

backout requeue name
This attribute is read-only. This is the excessive backout requeue name.

backout threshold
This attribute is read-only. This is the backout threshold.

base queue name
This attribute is read-only. This is the name of the queue that the alias resolves to.

creation date
This attribute is read-only. This is the queue creation data.

creation time
This attribute is read-only. This is the queue creation time.

current depth
This attribute is read-only. This is the number of messages on the queue.

default input open option
This attribute is read-only. This is the default open-for-input option.

default persistence
This attribute is read-only. This is the default message persistence.

default priority
This attribute is read-only. This is the default message priority.

definition type
This attribute is read-only. This is the queue definition type.

depth high event
This attribute is read-only. This is the control attribute for queue depth high events.

depth high limit
This attribute is read-only. This is the high limit for the queue depth.

depth low event
This attribute is read-only. This is the control attribute for queue depth low events.

depth low limit
This attribute is read-only. This is the low limit for the queue depth.

depth maximum event
This attribute is read-only. This is the control attribute for queue depth maximum events.

distribution list reference
An optional reference to an ImqDistributionList that can be used to distribute messages to more than one queue, including this one. Initially null.

Note: When an ImqQueue object is opened, any open ImqDistributionList object that it references is automatically closed.

distribution lists
This attribute is read-only. This is the capability of a transmission queue to support distribution lists.

dynamic queue name
This is the dynamic queue name. The initial value is “AMQ.*” for all Personal Computer and UNIX platforms.

harden get backout
This attribute is read-only. This determines whether to harden the backout count.

inhibit get
This determines whether get operations are allowed. The initial value is dependent on the queue definition. Only valid for an alias or local queue.

inhibit put
This determines whether put operations are allowed. The initial value is dependent on the queue definition.

initiation queue name
This attribute is read-only. This is the name of the initiation queue.

maximum depth
This attribute is read-only. This is the maximum number of messages allowed on the queue.

maximum message length
This attribute is read-only. The maximum length for any message on this queue, which may be less than the maximum for any queue managed by the associated queue manager.

message delivery sequence
This attribute is read-only. This determines whether message priority is relevant.

next distributed queue
This is the next object of this class, in no particular order, having the same **distribution list reference** as this object. Initially zero.

open input count
This attribute is read-only. This is the number of ImqQueue objects that are open for input.

open output count
This attribute is read-only. This is the number of ImqQueue objects that are open for output.

previous distributed queue
This is the previous object of this class, in no particular order, having the same **distribution list reference** as this object. Initially zero.

process name
This attribute is read-only. This is the name of the process definition.

queue manager name
This is the name of the queue manager (possibly remote) where the queue actually resides. The queue manager named here should not be confused with the ImqObject **connection reference** which references the (local) queue manager providing a connection. Initially null.

queue type
This attribute is read-only. This is the queue type.

remote queue manager name
This attribute is read-only. This is the name of the remote queue manager.

remote queue name
This attribute is read-only. This is the name of the remote queue as known on the remote queue manager.

retention interval
This attribute is read-only. This is the queue retention interval.

scope This attribute is read-only. This is the scope of the queue definition.

service interval
This attribute is read-only. This is the service interval.

service interval event
This attribute is read-only. This is the control attribute for service interval events.

shareability
This attribute is read-only. This determines whether the queue can be shared.

transmission queue name
This attribute is read-only. This is the name of the transmission queue.

trigger control
This is the trigger control. The initial value depends on the queue definition. Only valid for a local queue.

trigger data
This is the trigger data. The initial value depends on the queue definition. Only valid for a local queue.

trigger depth
This is the trigger depth. The initial value depends on the queue definition. Only valid for a local queue.

trigger message priority
This is the threshold message priority for triggers. The initial value depends on the queue definition. Only valid for a local queue.

trigger type
This is the trigger type. The initial value depends on the queue definition. Only valid for a local queue.

usage This attribute is read-only. This is the usage.

Constructors

ImqQueue( );
The default constructor.

ImqQueue(const ImqQueue & *queue* **);**
The copy constructor. The ImqObject **open status** will be FALSE.

ImqQueue(const char * *name* **);**
Sets the ImqObject **name**.

Object methods (public)

void operator = (const ImqQueue & *queue* **);**
Performs a close if necessary, and then copies instance data from *queue*. The ImqObject **open status** will be FALSE.

ImqBoolean backoutRequeueName(ImqString & *name* **);**
Provides a copy of the **backout requeue name**. Returns TRUE if successful.

ImqString backoutRequeueName( );
Returns the **backout requeue name** without any indication of possible errors.

ImqBoolean backoutThreshold(MQLONG & *threshold* **);**
Provides a copy of the **backout threshold**. Returns TRUE if successful.

MQLONG backoutThreshold( );
Returns the **backout threshold** value without any indication of possible errors.

ImqBoolean baseQueueName(ImqString & *name* **);**
Provides a copy of the **base queue name**. Returns TRUE if successful.

ImqString baseQueueName( );
Returns the **base queue name** without any indication of possible errors.

ImqBoolean creationDate(ImqString & *date* **);**
Provides a copy of the **creation date**. Returns TRUE if successful.

ImqString creationDate( );
Returns the **creation date** without any indication of possible errors.

ImqBoolean creationTime(ImqString & *time* **);**
Provides a copy of the **creation time**. Returns TRUE if successful.

ImqString creationTime( );
Returns the **creation time** without any indication of possible errors.

ImqBoolean currentDepth(MQLONG & *depth* **);**
Provides a copy of the **current depth**. Returns TRUE if successful.

MQLONG currentDepth( );
Returns the **current depth** without any indication of possible errors.

ImqBoolean defaultInputOpenOption(MQLONG & *option* **);**
Provides a copy of the **default input open option**. Returns TRUE if successful.

MQLONG defaultInputOpenOption( );
Returns the **default input open option** without any indication of possible errors.

ImqBoolean defaultPersistence(MQLONG & *persistence* **);**
Provides a copy of the **default persistence**. Returns TRUE if successful.

MQLONG defaultPersistence( );
Returns the **default persistence** without any indication of possible errors.

ImqBoolean defaultPriority(MQLONG & *priority* **);**
Provides a copy of the **default priority**. Returns TRUE if successful.

MQLONG defaultPriority( );
Returns the **default priority** without any indication of possible errors.

ImqBoolean definitionType(MQLONG & *type* **);**
Provides a copy of the **definition type**. Returns TRUE if successful.

MQLONG definitionType( );
Returns the **definition type** without any indication of possible errors.

ImqBoolean depthHighEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **depth high event**. Returns TRUE if successful.

MQLONG depthHighEvent( );
Returns the enablement state of the **depth high event** without any indication of possible errors.

ImqBoolean depthHighLimit(MQLONG & *limit* **);**
Provides a copy of the **depth high limit**. Returns TRUE if successful.

MQLONG depthHighLimit( );
Returns the **depth high limit** value without any indication of possible errors.

ImqBoolean depthLowEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **depth low event**. Returns TRUE if successful.

MQLONG depthLowEvent( );
Returns the enablement state of the **depth low event** without any indication of possible errors.

ImqBoolean depthLowLimit(MQLONG & *limit* **);**
Provides a copy of the **depth low limit**. Returns TRUE if successful.

MQLONG depthLowLimit( );
Returns the **depth low limit** value without any indication of possible errors.

ImqBoolean depthMaximumEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **depth maximum event**. Returns TRUE if successful.

MQLONG depthMaximumEvent( );
Returns the enablement state of the **depth maximum event** without any indication of possible errors.

ImqDistributionList * distributionListReference( ) const ;
Returns the **distribution list reference**.

void setDistributionListReference(ImqDistributionList & *list* **);**
Sets the **distribution list reference**.

void setDistributionListReference(ImqDistributionList * *list* **= 0);**
Sets the **distribution list reference**.

ImqBoolean distributionLists(MQLONG & *support* **);**
Provides a copy of the **distribution lists** value. Returns TRUE if successful.

MQLONG distributionLists( );
Returns the **distribution lists** value without any indication of possible errors.

ImqBoolean setDistributionLists(const MQLONG *support* **);**
Sets the **distribution lists** value. Returns TRUE if successful.

ImqString dynamicQueueName( ) const ;
Returns a copy of the **dynamic queue name**.

ImqBoolean setDynamicQueueName(const char * *name* **);**
Sets the **dynamic queue name**. The **dynamic queue name** can only be set while the ImqObject **open status** is FALSE. Returns TRUE if successful.

ImqBoolean get(ImqMessage & *msg***, ImqGetMessageOptions &** *options* **);**
| Retrieves a message from the queue, using the specified *options*. The
| ImqObject **openFor** method is invoked if necessary to ensure that the
| ImqObject **open options** include either (a) one of the MQOO_INPUT_*
| values, or (b) the MQOO_BROWSE value, depending on the *options*. If
| the *msg* object has an ImqCache **automatic buffer**, then the buffer will
| grow to accommodate any message retrieved. The **clearMessage** method is invoked against the *msg* object prior to retrieval. Returns TRUE if successful.

| **Note:** The result of the method invocation is FALSE if the ImqObject
| **reason code** is MQRC_TRUNCATED_MSG_FAILED, even
| though this **reason code** is classified as a warning. If a truncated message is accepted, then the ImqCache **message**
| **length** reflects the truncated length. In either event, the
| ImqMessage **total message length** indicates the number of
| bytes that were available.

ImqBoolean get(ImqMessage & *msg* **);**
As for the above method, except that default get message options are used.

ImqBoolean get(ImqMessage & *msg***, ImqGetMessageOptions &** *options***, const size_t** *buffer-size* **);**
As for the above methods, except that an overriding *buffer-size* is indicated. If the *msg* object employs an ImqCache **automatic buffer**, then the **resizeBuffer** method is invoked on the *msg* object prior to message retrieval, and the buffer will not grow further to accommodate any larger message.

ImqBoolean get(ImqMessage & *msg***, const size_t** *buffer-size* **);**
As for the above method, except that default get message options are used.

ImqBoolean hardenGetBackout(MQLONG & *harden* **);**
Provides a copy of the **harden get backout** value. Returns TRUE if successful.

MQLONG hardenGetBackout( );
Returns the **harden get backout** value without any indication of possible errors.

ImqBoolean inhibitGet(MQLONG & *inhibit* **);**
Provides a copy of the **inhibit get** value. Returns TRUE if successful.

MQLONG inhibitGet( );
Returns the **inhibit get** value without any indication of possible errors.

ImqBoolean setInhibitGet(const MQLONG *inhibit* **);**
Sets the **inhibit get** value. Returns TRUE if successful.

ImqBoolean inhibitPut(MQLONG & *inhibit* **);**
Provides a copy of the **inhibit put** value. Returns TRUE if successful.

MQLONG inhibitPut( );
Returns the **inhibit put** value without any indication of possible errors.

ImqBoolean setInhibitPut(const MQLONG *inhibit* **);**
Sets the **inhibit put** value. Returns TRUE if successful.

ImqBoolean initiationQueueName(ImqString & *name* **);**
Provides a copy of the **initiation queue name**. Returns TRUE if successful.

ImqString initiationQueueName( );
Returns the **initiation queue name** without any indication of possible errors.

ImqBoolean maximumDepth(MQLONG & *depth* **);**
Provides a copy of the **maximum depth**. Returns TRUE if successful.

MQLONG maximumDepth( );
Returns the **maximum depth** without any indication of possible errors.

ImqBoolean maximumMessageLength(MQLONG & *length* **);**
Provides a copy of the **maximum message length**. Returns TRUE if successful.

MQLONG maximumMessageLength( );
Returns the **maximum message length** without any indication of possible errors.

ImqBoolean messageDeliverySequence(MQLONG & *sequence* **);**
Provides a copy of the **message delivery sequence**. Returns TRUE if successful.

MQLONG messageDeliverySequence( );
Returns the **message delivery sequence** value without any indication of possible errors.

ImqQueue * nextDistributedQueue( ) const ;
Returns the **next distributed queue**.

ImqBoolean openInputCount(MQLONG & *count* **);**
Provides a copy of the **open input count**. Returns TRUE if successful.

MQLONG openInputCount( );
Returns the **open input count** without any indication of possible errors.

ImqBoolean openOutputCount(MQLONG & *count* **);**
Provides a copy of the **open output count**. Returns TRUE if successful.

MQLONG openOutputCount( );
Returns the **open output count** without any indication of possible errors.

ImqQueue * previousDistributedQueue( ) const ;
Returns the **previous distributed queue**.

ImqBoolean processName(ImqString & *name* **);**
Provides a copy of the **process name**. Returns TRUE if successful.

ImqString processName( );
Returns the **process name** without any indication of possible errors.

ImqBoolean put(ImqMessage & *msg* **);**
Places a message onto the queue, using default put message options. Uses the ImqObject **openFor** method if necessary to ensure that the ImqObject **open options** include MQOO_OUTPUT. Returns TRUE if successful.

ImqBoolean put(ImqMessage & *msg***, ImqPutMessageOptions &** *pmo* **);**
Places a message onto the queue, using the specified *pmo*. Uses the ImqObject **openFor** method as necessary to ensure that the ImqObject **open options** include MQOO_OUTPUT, and (if the *pmo* **options** include any of MQPMO_PASS_IDENTITY_CONTEXT, MQPMO_PASS_ALL_CONTEXT, MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT) corresponding MQOO_*_CONTEXT values. Returns TRUE if successful.

Note: If the *pmo* includes a **context reference**, then the referenced object will be opened if necessary to provide a context.

ImqString queueManagerName( ) const ;
Returns the **queue manager name**.

ImqBoolean setQueueManagerName(const char * *name* **);**
Sets the **queue manager name**. The **queue manager name** can only be set while the ImqObject **open status** is FALSE. Returns TRUE if successful.

ImqBoolean queueType(MQLONG & *type* **);**
Provides a copy of the **queue type** value. Returns TRUE if successful.

MQLONG queueType( );
Returns the **queue type** without any indication of possible errors.

ImqBoolean remoteQueueManagerName(ImqString & *name* **);**
Provides a copy of the **remote queue manager name**. Returns TRUE if successful.

ImqString remoteQueueManagerName( );
Returns the **remote queue manager name** without any indication of possible errors.

ImqBoolean remoteQueueName(ImqString & *name* **);**
Provides a copy of the **remote queue name**. Returns TRUE if successful.

ImqString remoteQueueName( );
Returns the **remote queue name** without any indication of possible errors.

ImqBoolean retentionInterval(MQLONG & *interval* **);**
Provides a copy of the **retention interval**. Returns TRUE if successful.

MQLONG retentionInterval( );
Returns the **retention interval** without any indication of possible errors.

ImqBoolean scope(MQLONG & *scope* **);**
Provides a copy of the **scope**. Returns TRUE if successful.

MQLONG scope( );
Returns the **scope** without any indication of possible errors.

ImqBoolean serviceInterval(MQLONG & *interval* **);**
Provides a copy of the **service interval**. Returns TRUE if successful.

MQLONG serviceInterval( );
Returns the **service interval** without any indication of possible errors.

ImqBoolean serviceIntervalEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **service interval event**. Returns TRUE if successful.

MQLONG serviceIntervalEvent( );
Returns the anablement state of the **service interval event** without any indication of possible errors.

ImqBoolean shareability(MQLONG & *shareability* **);**
Provides a copy of the **shareability** value. Returns TRUE if successful.

MQLONG shareability( );
Returns the **shareability** value without any indication of possible errors.

ImqBoolean transmissionQueueName(ImqString & *name* **);**
Provides a copy of the **transmission queue name**. Returns TRUE if successful.

ImqString transmissionQueueName( );
Returns the **transmission queue name** without any indication of possible errors.

ImqBoolean triggerControl(MQLONG & *control* **);**
Provides a copy of the **trigger control** value. Returns TRUE if successful.

MQLONG triggerControl( );
Returns the **trigger control** value without any indication of possible errors.

ImqBoolean setTriggerControl(const MQLONG *control* **);**
Sets the **trigger control** value. Returns TRUE if successful.

ImqBoolean triggerData(ImqString & *data* **);**
Provides a copy of the **trigger data**. Returns TRUE if successful.

ImqString triggerData( );
Returns a copy of the **trigger data** without any indication of possible errors.

ImqBoolean setTriggerData(const char * *data* **);**
Sets the **trigger data**. Returns TRUE if successful.

ImqBoolean triggerDepth(MQLONG & *depth* **);**
Provides a copy of the **trigger depth**. Returns TRUE if successful.

MQLONG triggerDepth( );
Returns the **trigger depth** without any indication of possible errors.

ImqBoolean setTriggerDepth(const MQLONG *depth* **);**
Sets the **trigger depth**. Returns TRUE if successful.

ImqBoolean triggerMessagePriority(MQLONG & *priority* **);**
Provides a copy of the **trigger message priority**. Returns TRUE if successful.

MQLONG triggerMessagePriority( );
Returns the **trigger message priority** without any indication of possible errors.

ImqBoolean setTriggerMessagePriority(const MQLONG *priority* **);**
Sets the **trigger message priority**. Returns TRUE if successful.

ImqBoolean triggerType(MQLONG & *type* **);**
Provides a copy of the **trigger type**. Returns TRUE if successful.

MQLONG triggerType( );
Returns the **trigger type** without any indication of possible errors.

ImqBoolean setTriggerType(const MQLONG *type* **);**
Sets the **trigger type**. Returns TRUE if successful.

ImqBoolean usage(MQLONG & *usage* **);**
Provides a copy of the **usage** value. Returns TRUE if successful.

MQLONG usage( );
Returns the **usage** value without any indication of possible errors.

Object methods (protected)

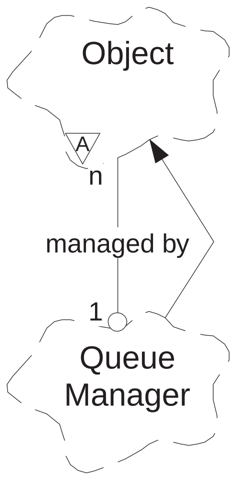
void setNextDistributedQueue(ImqQueue * *queue* **= 0);**
Sets the **next distributed queue**.

void setPreviousDistributedQueue(ImqQueue * *queue* **= 0);**
Sets the **previous distributed queue**.

Reason codes

MQRC_CONTEXT_OBJECT_NOT_VALID
MQRC_CONTEXT_OPEN_ERROR
MQRC_CURSOR_NOT_VALID
MQRC_NO_BUFFER
MQRC_REOPEN_EXCL_INPUT_ERROR
MQRC_REOPEN_INQUIRE_ERROR
MQRC_REOPEN_SAVED_CONTEXT_ERR
MQRC_REOPEN_TEMPORARY_Q_ERROR
(reason codes from MQGET)
(reason codes from MQPUT)

ImqQueueManager



This class encapsulates a queue manager (an MQSeries object or type MQOT_Q_MGR).

Other relevant classes

ImqObject (see "ImqObject" on page 53)

Object attributes

authority event
This attribute is read-only. This controls authority events.

begin options
These are the options that apply to the **begin** method. Initially MQBO_NONE.

character set
This attribute is read-only. This is the coded character set identifier.

command input queue name
This attribute is read-only. This is the system command input queue name.

command level
This attribute is read-only. This is the command level supported by the queue manager.

connect options
These are the options that apply to the **connect** method. Initially MQCNO_NONE.

connection status
This attribute is read-only. This is TRUE when connected to the queue manager.

dead-letter queue name
This attribute is read-only. This is the name of the dead-letter queue.

default transmission queue name
This attribute is read-only. This is the default transmission queue name.

distribution lists
This attribute is read-only. This is the capability of the queue manager to support distribution lists.

first managed object
The first of one or more objects of class ImqObject, in no particular order, in which the ImqObject **connection reference** addresses this object. Initially zero.

inhibit event
This attribute is read-only. This controls inhibit events.

local event
This attribute is read-only. This controls local events.

maximum handles
This attribute is read-only. This is the maximum number of handles.

maximum message length
This attribute is read-only. This is the maximum possible length for any message on any queue managed by this queue manager.

maximum priority
This attribute is read-only. This is the maximum message priority.

maximum uncommitted messages
This attribute is read-only. This is the maximum number of uncommitted messages within a unit or work.

performance event
This attribute is read-only. This controls performance events.

platform This attribute is read-only. This is the platform on which the queue manager resides.

remote event
This attribute is read-only. This controls remote events.

start-stop event
This attribute is read-only. This controls start-stop events.

sync-point availability
| This attribute is read-only. This is the availability[3] of sync-point
| participation.

trigger interval
This attribute is read-only. This is the trigger interval.

Constructors

ImqQueueManager( );
The default constructor.

ImqQueueManager(const ImqQueueManager & *manager* **);**
The copy constructor. The **connection status** will be FALSE.

| 3 Although the **begin**, **backout** and **commit** methods will all fail with MQRC_ENVIRONMENT_ERROR on the AS/400 platform,
| sync-point can be programmed using the "_Rcommit" and "_Rback" native system calls. Starting a unit of work is achieved by
| starting the MQSeries application program under commitment control using the STRCMTCTL command. See "Syncpoints in
| MQSeries for AS/400 applications" of the MQSeries Application Programming Guide for further details.

ImqQueueManager(const char * *name* **);**
Sets the ImqObject **name** to *name*.

Destructors

When an ImqQueueManager object is destroyed, it is automatically disconnected.

Object methods (public)

void operator = (const ImqQueueManager & *mgr* **);**
Disconnects if necessary, and then copies instance data from *mgr*. The **connection status** will be FALSE.

ImqBoolean authorityEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **authority event**. Returns TRUE if successful.

MQLONG authorityEvent( );
Returns the enablement state of the **authority event** without any indication of possible errors.

ImqBoolean backout( );
Backs out uncommitted changes. Returns TRUE if successful.

ImqBoolean begin( );
Begins a unit of work. The **begin options** affect the behavior of this method. Returns TRUE if successful.

MQLONG beginOptions( ) const ;
Returns the **begin options**.

void setBeginOptions(const MQLONG *options* **= MQBO_NONE);**
Sets the **begin options**.

ImqBoolean characterSet(MQLONG & *ccsid* **);**
Provides a copy of the **character set**. Returns TRUE if successful.

MQLONG characterSet( );
Returns a copy of the **character set**, without any indication of possible errors.

ImqBoolean commandInputQueueName(ImqString & *name* **);**
Provides a copy of the **command input queue name**. Returns TRUE if successful.

ImqString commandInputQueueName( );
Returns the **command input queue name** without any indication of possible errors.

ImqBoolean commandLevel(MQLONG & *level* **);**
Provides a copy of the **command level**. Returns TRUE if successful.

MQLONG commandLevel( );
Returns the **command level** without any indication of possible errors.

ImqBoolean commit( );
Commits uncommitted changes. Returns TRUE if successful.

ImqBoolean connect( );
Connects to the queue manager with the given ImqObject **name**, the default being the local queue manager. Use the ImqObject **setName** method before connection if you wish to connect to a specific queue

manager. The **connect options** affect the behavior of this method. Sets the **connection status** to TRUE. Returns TRUE if successful.

Note: More than one ImqQueueManager object can be connected to the same queue manager, and all will use the same MQHCONN

MQLONG connectOptions( ) const ;
Returns the **connect options**.

void setConnectOptions(const MQLONG *options* **= MQCNO_NONE);**
Sets the **connect options**.

ImqBoolean connectionStatus( ) const ;
Returns the **connection status**.

ImqBoolean deadLetterQueueName(ImqString & *name* **);**
Provides a copy of the **dead-letter queue name**. Returns TRUE if successful.

ImqString deadLetterQueueName( );
Returns a copy of the **dead-letter queue name**, without any indication of possible errors.

ImqBoolean defaultTransmissionQueueName(ImqString & *name* **);**
Provides a copy of the **default transmission queue name**. Returns TRUE if successful.

ImqString defaultTransmissionQueueName( );
Returns the **default transmission queue name** without any indication of possible errors.

ImqBoolean disconnect( );
Disconnects from the queue manager and sets the **connection status** to FALSE. All ImqProcess and ImqQueue objects associated with this object are closed and their **connection reference** severed prior to disconnection. If more than one ImqQueueManager object is connected to the same queue manager, then only the last to disconnect will perform a physical disconnection; others will perform a logical disconnection. Uncommitted changes are committed (on physical disconnection only). Returns TRUE if successful.

ImqBoolean distributionLists(MQLONG & *support* **);**
Provides a copy of the **distribution lists** value. Returns TRUE if successful.

MQLONG distributionLists( );
Returns the **distribution lists** value without any indication of possible errors.

ImqObject * firstManagedObject( ) const ;
Returns the **first managed object**.

ImqBoolean inhibitEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **inhibit event**. Returns TRUE if successful.

MQLONG inhibitEvent( );
Returns the enablement state of the **inhibit event** without any indication of possible errors.

ImqBoolean localEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **local event**. Returns TRUE if successful.

MQLONG localEvent( );
Returns the enablement state of the **local event** without any indication of possible errors.

ImqBoolean maximumHandles(MQLONG & *number* **);**
Provides a copy of the **maximum handles**. Returns TRUE if successful.

MQLONG maximumHandles( );
Returns the **maximum handles** without any indication of possible errors.

ImqBoolean maximumMessageLength(MQLONG & *length* **);**
Provides a copy of the **maximum message length**. Returns TRUE if successful.

MQLONG maximumMessageLength( );
Returns the **maximum message length** without any indication of possible errors.

ImqBoolean maximumPriority(MQLONG & *priority* **);**
Provides a copy of the **maximum priority**. Returns TRUE if successful.

MQLONG maximumPriority( );
Returns a copy of the **maximum priority**, without any indication of possible errors.

ImqBoolean maximumUncommittedMessages(MQLONG & *number* **);**
Provides a copy of the **maximum uncommitted messages**. Returns TRUE if successful.

MQLONG maximumUncommittedMessages( );
Returns the **maximum uncommitted messages** without any indication of possible errors.

ImqBoolean performanceEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **performance event**. Returns TRUE if successful.

MQLONG performanceEvent( );
Returns the enablement state of the **performance event** without any indication of possible errors.

ImqBoolean platform(MQLONG & *platform* **);**
Provides a copy of the **platform**. Returns TRUE if successful.

MQLONG platform( );
Returns the **platform** without any indication of possible errors.

ImqBoolean remoteEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **remote event**. Returns TRUE if successful.

MQLONG remoteEvent( );
Returns the enablement state of the **remote event** without any indication of possible errors.

ImqBoolean startStopEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **start-stop event**. Returns TRUE if successful.

MQLONG startStopEvent( );
Returns the enablement state of the **start-stop event** without any indication of possible errors.

ImqBoolean syncPointAvailability(MQLONG & *sync* **);**
Provides a copy of the **sync-point availability** value. Returns TRUE if successful.

MQLONG syncPointAvailability( );
Returns a copy of the **sync-point availability** value, without any indication of possible errors.

ImqBoolean triggerInterval(MQLONG & *interval* **);**
Provides a copy of the **trigger interval**. Returns TRUE if successful.

MQLONG triggerInterval( );
Returns the **trigger interval** without any indication of possible errors.

Object methods (protected)

void setFirstManagedObject(const ImqObject * *object* **= 0);**
Sets the **first managed object**.

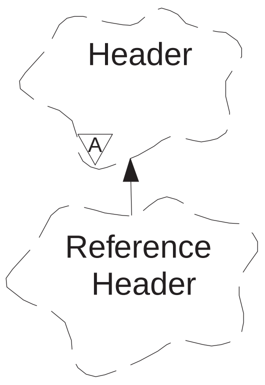
Object data (protected)

MQHCONN *ohconn*
The MQSeries connection handle (only meaningful while the **connection status** is TRUE).

Reason codes

(reason codes for MQBACK)
(reason codes for MQBEGIN)
(reason codes for MQCMIT)
(reason codes for MQCONNX)
(reason codes for MQDISC)

ImqReferenceHeader



This class encapsulates specific features of the MQRMH data structure.

Other relevant classes

ImqBinary (see “ImqBinary” on page 23)
ImqHeader (see “ImqHeader” on page 38)
ImqItem (see “ImqItem” on page 43)
ImqMessage (see “ImqMessage” on page 45)
ImqString (see “ImqString” on page 82)

Object attributes

destination environment
This is the environment for the destination. Initially a null string.

destination name
This is the name of the data destination. Initially a null string.

instance id
This is a binary value (MQBYTE24) of length MQ_OBJECT_INSTANCE_ID_LENGTH. The initial value is MQOII_NONE.

logical length
This is the logical, or intended, length of message data that follows this header. Initially zero.

logical offset
This is a logical offset for the message data that follows, to be interpreted in the context of the data as a whole, at the ultimate destination. Initially zero.

logical offset 2
This is a high-order extension to the **logical offset**. Initially zero.

reference type
This is the reference type. Initially a null string.

source environment
This is the environment for the source. Initially a null string.

source name
This is the name of the data source. Initially a null string.

Constructors

ImqReferenceHeader( );
The default constructor.

ImqReferenceHeader(const ImqReferenceHeader & *header* **);**
The copy constructor.

Overloaded “ImqItem” methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Inserts an MQRMH data structure into the message buffer at the beginning, moving existing message data further along. Sets the *msg* **format** to MQFMT_REF_MSG_HEADER.

See the ImqHeader class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Reads an MQRMH data structure from the message buffer.

To be successful, the ImqMessage **format** must be MQFMT_REF_MSG_HEADER.

See the ImqHeader class method description for further details.

Object methods (public)

void operator = (const ImqReferenceHeader & *header* **);**
Instance data is copied from *header*, replacing the existing instance data.

ImqString destinationEnvironment( ) const ;
Returns a copy of the **destination environment**.

void setDestinationEnvironment(const char * *environment* **= 0);**
Sets the **destination environment**.

ImqString destinationName( ) const ;
Returns a copy of the **destination name**.

void setDestinationName(const char * *name* **= 0);**
Sets the **destination name**.

ImqBinary instanceId( ) const ;
Returns a copy of the **instance id**.

ImqBoolean setInstanceId(const ImqBinary & *id* **);**
Sets the **instance id**. The **data length** of *token* must be either 0 or MQ_OBJECT_INSTANCE_ID_LENGTH. Returns TRUE if successful.

void setInstanceId(const MQBYTE24 *id* **= 0);**
Sets the **instance id**. *id* may be zero, which is the same as specifying MQOII_NONE. If *id* is non-zero, then it must address MQ_OBJECT_INSTANCE_ID_LENGTH bytes of binary data. When using pre-defined values such as MQOII_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQOII_NONE.

MQLONG logicalLength( ) const ;
Returns the **logical length**.

void setLogicalLength(const MQLONG *length* **);**
Sets the **logical length**.

MQLONG logicalOffset( ) const ;
Returns the **logical offset**.

void setLogicalOffset(const MQLONG *offset* **);**
Sets the **logical offset**.

MQLONG logicalOffset2( ) const ;
Returns the **logical offset 2**.

void setLogicalOffset2(const MQLONG *offset* **);**
Sets the **logical offset 2**.

ImqString referenceType( ) const ;
Returns a copy of the **reference type**.

void setReferenceType(const char * *name* **= 0);**
Sets the **reference type**.

ImqString sourceEnvironment( ) const ;
Returns a copy of the **source environment**.

void setSourceEnvironment(const char * *environment* **= 0);**
Sets the **source environment**.

ImqString sourceName( ) const ;
Returns a copy of the **source name**.

void setSourceName(const char * *name* **= 0);**
Sets the **source name**.

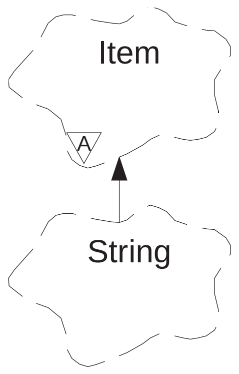
Object data (protected)

MQRMH *omqrmh*
The MQRMH data structure.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR
MQRC_STRUC_LENGTH_ERROR

ImqString



This class provides character string storage and manipulation for null-terminated strings. An ImqString can be used in place of a **char *** in most situations where a parameter calls for a **char ***.

Other relevant classes

ImqItem (see "ImqItem" on page 43)
ImqMessage (see "ImqMessage" on page 45)

Object attributes

characters	Those characters in the **storage** which precede a trailing null.
length	The number of bytes in the **characters**. If there is no **storage**, then the **length** is zero. Initially zero.
storage	A volatile array of bytes of arbitrary size. A trailing null must always be present in the **storage** after the **characters**, so that the end of the **characters** can be detected. Methods ensure that this situation is maintained, but care must be taken, when setting bytes in the array directly, to ensure that a trailing null exists after modification. Initially there is no **storage**.

Constructors

ImqString( );
The default constructor.

ImqString(const ImqString & *string* **);**
The copy constructor.

ImqString(const char *c* **);**
The **characters** comprise *c*.

ImqString(const char * *text* **);**
The **characters** are copied from *text*.

ImqString(const void * *buffer***, const size_t** *length* **);**
Copies *length* bytes starting from *buffer* and assigns them to the **characters**. Substitution is made for any null characters copied. The substitution character is a period (.). No special consideration is given to any other non-printable or non-displayable characters copied.

Class methods (public)

static ImqBoolean copy(char * *destination-buffer*, **const size_t** *length*, **const char *** *source-buffer*, **const char** *pad* **= 0);**

Copies up to *length* bytes from *source-buffer* to *destination-buffer*. If the number of characters in *source-buffer* is insufficient, then the remaining space in *destination-buffer* is filled with *pad* characters. *source-buffer* may be zero. *destination-buffer* may be zero if *length* is also zero. Returns TRUE if successful.

Overloaded “ImqItem” methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**

Copies the **characters** to the message buffer, replacing any existing content. Sets the *msg* **format** to MQFMT_STRING.

See the parent class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**

Sets the **characters** by transferring the remaining data from the message buffer, replacing the existing **characters**.

To be successful, the **encoding** of the *msg* object should be MQENC_NATIVE. It is recommended that messages be retrieved with MQGMO_CONVERT to MQENC_NATIVE.

To be successful, the ImqMessage **format** must be MQFMT_STRING.

See the parent class method description for further details.

Object methods (public)

char & operator [ ] (const size_t *offset* **) const ;**

References the character at offset *offset* in the **storage**. It is the user’s responsibility to ensure that the relevant byte exists and is addressable.

ImqString operator ( ) (const size_t *offset*, **const size_t** *length* **= 1) const ;**

Returns a sub-string by copying bytes from the **characters** starting at *offset*. If *length* is zero, then the rest of the **characters** are returned. If the combination of *offset* and *length* does not produce a reference within the **characters**, then an empty ImqString is returned.

void operator = (const ImqString & *string* **);**

Instance data is copied from *string*, replacing the existing instance data.

ImqString operator + (const char *c* **) const ;**

Returns the result of appending *c* to the **characters**.

ImqString operator + (const char * *text* **) const ;**

Returns the result of appending *text* to the **characters**. This may also be inverted. For example:

```
strOne + “string two”[4] ;
“string one” + strTwo ;
```

ImqString operator + (const ImqString & *string1* **) const ;**

Returns the result of appending *string1* to the **characters**.

4 Although most compilers accept **strOne + “string two”;** Microsoft Visual C++ requires **strOne + (char *)“string two” ;**

ImqString operator + (const double *number* **) const ;**
Returns the result of appending *number* to the **characters** after conversion to text.

ImqString operator + (const long *number* **) const ;**
Returns the result of appending *number* to the **characters** after conversion to text.

void operator += (const char *c* **);**
c is appended to the **characters**.

void operator += (const char * *text* **);**
Appends *text* to the **characters**.

void operator += (const ImqString & *string* **);**
Appends *string* to the **characters**.

void operator += (const double *number* **);**
Appends *number* to the **characters** after conversion to text.

void operator += (const long *number* **);**
Appends *number* to the **characters** after conversion to text.

void operator char * ( ) const ;
Returns the address of the first byte in the **storage**. May be zero, and is volatile.

ImqBoolean operator < (const ImqString & *string* **) const ;**
ImqBoolean operator > (const ImqString & *string* **) const ;**
ImqBoolean operator <= (const ImqString & *string* **) const ;**
ImqBoolean operator >= (const ImqString & *string* **) const ;**
ImqBoolean operator == (const ImqString & *string* **) const ;**
ImqBoolean operator != (const ImqString & *string* **) const ;**
Compares the **characters** with those of *string* using the **compare** method. Returns either TRUE or FALSE.

short compare(const ImqString & *string* **) const ;**
Compares the **characters** with those of *string*. The result is zero if the **characters** are equal, negative if “less than” and positive if “greater than”. Comparison is case-sensitive. A null ImqString is regarded as “less than” a non-null ImqString.

ImqBoolean copyOut(char * *buffer*, **const size_t** *length*, **const char** *pad* **= 0);**
Copies up to *length* bytes from the **characters** to the *buffer*. If the number of **characters** is insufficient, then the remaining space in *buffer* is filled with *pad* characters. *buffer* may be zero if *length* is also zero. Returns TRUE if successful.

size_t copyOut(long & *number* **) const ;**
Sets *number* from the **characters** after conversion from text. Returns the number of characters involved in the conversion. If this is zero, then no conversion has been performed and *number* is not set. A convertible character sequence must begin with:

```
<blank(s)>
<+|->
digit(s)
```

Figure 11. Format for string text to integer conversion

size_t copyOut(ImqString & *token*, **const char** *c* **= ‘ ’) const ;**

If the **characters** contain one or more characters different to *c*, then a token is identified as the first contiguous sequence of such characters. In this case *token* is set to that sequence, and the value returned is the sum of the number of leading characters *c* and the number of bytes in the sequence. Otherwise, zero is returned and *token* is not set.

size_t cutOut(long & *number* **);**

Sets *number* as for the **copy** method, but also removes from **characters** the number of bytes indicated by the return value. For example, the following string may be cut into three numbers by using **cutOut**(*number*) three times:

```
strNumbers = “-1 0     +55 ”;

while ( strNumbers.cutOut( number ) );
number becomes -1, then 0, then 55
leaving strNumbers == “ ”
```

Figure 12. Retrieving integers from string text

size_t cutOut(ImqString & *token*, **const char** *c* **= ‘ ’);**

Sets *token* as for the **copyOut** method, and removes from **characters** the *strToken* characters and also any characters *c* which preceed the *token* characters. If *c* is not a blank, then characters *c* which directly succeed the *token* characters are also removed. Returns the number of characters removed. For example, the following string may be cut into three tokens by using **cutOut**(*token*) three times:

```
strText = “  Program Version 1.1  ”;

while ( strText.cutOut( token ) );

// token becomes “Program”, then “Version”,
// then “1.1” leaving strText == “  ”
```

Figure 13. Retrieving tokens from string text

Another example shows how a DOS path name might be parsed as follows:

```
strPath = “C:\OS2\BITMAP\OS2LOGO.BMP”

strPath.cutOut( strDrive, ':' );
strPath.stripLeading( ':' );
while ( strPath.cutOut( strFile, '\' ) );

// strDrive becomes “A”.
// strFile becomes “OS2”, then “BITMAP”,
// then “OS2LOGO.BMP” leaving strPath empty.
```

Figure 14. Parsing a path in a string

ImqBoolean find(const ImqString & *string* **);**
Searches for an exact match for *string* anywhere within the **characters**. If no match is found, returns FALSE. Otherwise, returns TRUE. If *string* is null, returns TRUE.

ImqBoolean find(const ImqString & *string*, **size_t &** *offset* **);**
Searches for an exact match for *string* somewhere within the **characters** from offset *offset* onwards. If *string* is null, returns TRUE without updating *offset*. If no match is found, returns FALSE; note that the value of *offset* may have been increased. If a match is found, returns TRUE and updates *offset* to the offset of *string* within the **characters**.

size_t length( ) const ;
Returns the **length**.

ImqBoolean pasteIn(const double *number*, **const char *** *format* **= "%f");**
number is appended to the **characters** after conversion to text. Returns TRUE if successful.

The specification *format* is used to format the floating point conversion. If specified, it should be one suitable for use with **printf** and floating point numbers, for example **"%.3f"**.

ImqBoolean pasteIn(const long *number* **);**
number is appended to the **characters** after conversion to text. Returns TRUE if successful.

ImqBoolean pasteIn(const void * *buffer*, **const size_t** *length* **);**
Appends *length* bytes from *buffer* to the **characters**, and adds a final trailing null. A substitution is made for any null characters copied. The substitution character is a period (.). No special consideration is given to any other non-printable or non-displayable characters copied. Returns TRUE if successful.

ImqBoolean set(const char * *buffer*, **const size_t** *length* **);**
Sets the **characters** from a fixed-length character field, which may or may not contain a null. A null is appended to the characters from the fixed-length field if necessary. Returns TRUE if successful.

size_t storage( ) const ;
Returns the number of bytes in the **storage**.

ImqBoolean setStorage(const size_t *length* **);**
(Re)allocates the **storage** and returns the number of bytes currently allocated. Any original **characters**, including any trailing null, are preserved if there is still room for them, but any additional storage is not initialized.

Returns TRUE if successful.

size_t stripLeading(const char *c* **= ' ');**
Strips leading characters *c* from the **characters** and returns the number removed.

size_t stripTrailing(const char *c* **= ' ');**
Strips trailing characters *c* from the **characters** and returns the number removed.

ImqString upperCase( ) const ;
Returns an uppercase copy of the **characters**.

Object methods (protected)

ImqBoolean assign(const ImqString & *string* **);**

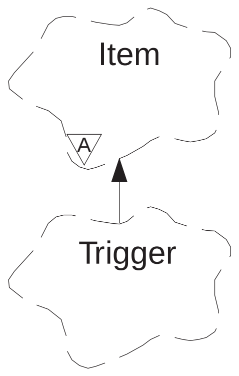
Equivalent to the equivalent **operator =** method, but non-virtual.

Returns TRUE if successful.

Reason codes

MQRC_DATA_TRUNCATED
MQRC_NULL_POINTER
MQRC_STORAGE_NOT_AVAILABLE

ImqTrigger



This class encapsulates the MQTM data structure (see Appendix B, “MQI cross-reference” on page 95). Objects of this class are typically used by a trigger monitor program, whose task is to wait for these particular messages and act on them to ensure that other MQSeries applications are started when messages are waiting for them.

See the IMQSTRG sample program for a usage example.

Other relevant classes

ImqGetMessageOptions (see “ImqGetMessageOptions” on page 35)
ImqItem (see “ImqItem” on page 43)
ImqMessage (see “ImqMessage” on page 45)
ImqString (see “ImqString” on page 82)

Object attributes

application id
: This is the identity of the application that sent the message. The initial value is a null string.

application type
: This is the type of application that sent the message. The initial value is zero.

environment data
: This is environment data for the process. The initial value is a null string.

process name
: This is the process name. The initial value is a null string.

queue name
: This is the name of the queue to be started. The initial value is a null string.

trigger data
: This is trigger data for the process. The initial value is a null string.

user data
: This is user data for the process. The initial value is a null string.

Constructors

ImqTrigger( );
The default constructor.

ImqTrigger(const ImqTrigger & *trigger* **);**
The copy constructor.

Overloaded “ImqItem” methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Writes an MQTM data structure to the message buffer, replacing any existing content. Sets the *msg* **format** to MQFMT_TRIGGER.

See the ImqItem class method description for further details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Reads an MQTM data structure from the message buffer.

To be successful, the ImqMessage **format** must be MQFMT_TRIGGER.

See the ImqItem class method description for further details.

Object methods (public)

void operator = (const ImqTrigger & *trigger* **);**
Instance data is copied from *trigger*, replacing the existing instance data.

ImqString applicationId( ) const ;
Returns a copy of the **application id**.

void setApplicationId(const char * *id* **);**
Sets the **application id**.

MQLONG applicationType( ) const ;
Returns the **application type**.

void setApplicationType(const MQLONG *type* **);**
Sets the **application type**.

ImqBoolean copyOut(MQTMC2 * *ptmc2* **);**
This class encapsulates the MQTM data structure which is the one received on initiation queues. This method fills in an equivalent MQTMC2 data structure provided by the caller, and sets the QMgrName field (which is not present in the MQTM data structure) to all blanks. The MQTMC2 data structure is traditionally used as a parameter to applications started by a trigger monitor. Returns TRUE if successful.

ImqString environmentData( ) const ;
Returns a copy of the **environment data**.

void setEnvironmentData(const char * *data* **);**
Sets the **environment data**.

ImqString processName( ) const ;
Returns a copy of the **process name**.

void setProcessName(const char * *name* **);**
Sets the **process name**.

ImqString queueName( ) const ;
Returns a copy of the **queue name**.

void setQueueName(const char * *name* **);**
Sets the **queue name**.

ImqString triggerData( ) const ;
Returns a copy of the **trigger data**.

void setTriggerData(const char * *data* **);**
Sets the **trigger data**.

ImqString userData( ) const ;
Returns a copy of the **user data**.

void setUserData(const char * *data* **);**
Sets the **user data**.

Object data (protected)

MQTM *omqtm*
The MQTM data structure.

Reason codes

MQRC_NULL_POINTER

Appendix A. Compiling and linking

| The compilers for each platform are listed in "Compilers for MQSeries platforms," together with the switches and link libraries to use.

| If you are writing programs for the AS/400 platform, see "Compiling C++ sample programs for the AS/400" on page 92.

| If you are writing programs for the Windows 95 and Windows NT platforms, see "Compiling VisualAge C++ sample programs for Windows 95 and NT" on page 94.

Compilers for MQSeries platforms

| The compilers can be used on both the MQSeries client and the MQSeries server, unless indicated otherwise in the table.

Table 3. MQSeries C++ switches and link libraries

Platform	Compiler	Switches	Libraries
AIX	IBM C Set++ Version 3.1 for AIX	xlC[_r] -qchars=signed -I/usr/lpp/mqm/inc	-limqb23ia[_r] -limq{c│s}23ia[_r]
AS/400 (server only)	IBM VisualAge for C++ for AS/400	iccas /C /J-	bndsrvpgm(qmqm/imqb23i4 qmqm/imqs23i4 qmqm/amqzstub)
HP-UX	HP C++ Version 3.1	CC -w	-limqb23ch[_r] -limq{c│s}23ch[_r]
OS/2	IBM VisualAge for C++ Version 3.0 for OS/2	icc /Gd /Gm /Gs /J-	imqb23i2 imq{c│s}23i2
Sun Solaris	Sun SPARCompiler C++ Release 4.1	CC -mt	-limqb23ss -limq{c│s}23ss {-lmqic│-lmqm} -lmqmcs -lmqmzse -lsocket -lnsl -ldl
Windows 3.1 (16-bit client only)	Microsoft Visual C++ Version 1.5 for Windows 3.1	cl -ALw	imqb23vw imqc23vw mqic
Windows 95, Windows NT	IBM VisualAge for C++ for Windows Version 3.5 for Windows 95 and NT	icc /Gd /Gm /Gs /J-	imqb23in imq{c│s}23in
Windows 95, Windows NT	Microsoft Visual C++ Version 4.0 for Windows 95 and NT	cl -MT	imqb23vn imq{c│s}23vn

Compiling C++ sample programs for the AS/400

This section is aimed at the C++ programmer who wishes to write programs that will run on the AS/400 platform.

There is no native AS/400 compiler for C++ programs. A cross-compiler is required that will produce an object module that can be linked by the AS/400 binder. VisualAge for C++ for AS/400 is the cross-compiler that runs on the OS/2 platform. Use of this cross-compiler allows a C++ programmer to use the rich graphical development environment of OS/2 to develop the program, whilst being able to build the AS/400 executable transparently on the target AS/400.

Setting up on the OS/2 platform

Set up the C++ development environment on the OS/2 platform as follows:

1. Install the VisualAge for C++ for OS/2 compiler, either from the VisualAge for C++ for OS/2 media, or from the target AS/400, if available[5].

   To install the compiler from the AS/400 machine you can use the IBM Client Access program to give access to the AS/400 shared directories. Move to directory `QDLS\QCTT\MRI2924\QCTTOS` and type `install`.

   Verify installation by compiling a sample application.

   Further details can be found in the *VisualAge for C++ for OS/2* manual.

2. Ensure that the VisualAge for C++ for AS/400 OS/2 Client is available[5] on the target AS/400.

   To install the cross-compiler from the AS/400 machine you can use the IBM Client Access program to give access to the AS/400 shared directories. Move to directory `QDLS\QCTT\MRI2924\QCTTAS` and type `install`.

   This results in a new folder on the OS/2 desktop entitled “VisualAge for C++ for AS/400”. This folder contains project templates, and help documentation specific to building AS/400 executables.

   Further details can be found in the *VisualAge for C++ for AS/400 User's Guide*.

   The above installation allows two modes of operation:

   - Fully automatic mode. The VisualAge C++ graphical front end is used to edit the source and compile the source into object code. The object code is transferred automatically to the target AS/400 to be linked into an AS/400 executable by the AS/400 binder.
   - Disconnected mode. The object code is left on the OS/2 platform as an intermediate file with a `.qwo` extension.

     You then have to transfer the intermediate object code across to the target AS/400, and invoke the AS/400 binder with appropriate options in order to produce an AS/400 native executable.

   Further details on the above modes, and AS/400 related restrictions, can be found in the “C++ User's Guide” in the “VisualAge for C++ for AS/400” folder on the OS/2 desktop.

[5] Availability can be checked by using the AS/400 command “go licpgm” and option 10 “display installed licensed programs”.

3. Install the MQSeries for AS/400 C++ toolkit for OS/2 onto the OS/2 platform as follows:

   Use the IBM Client Access/400 program to give access to the AS/400 shared directories. Move to `QDLS\QMQM\QIMQOS2\EN_US` (for US English) and type `install`.

   The above installation provides local copies of the MQSeries C++ and C header files, and the C++ sample programs, for use with the cross-compiler. The environment variable INCLUDE_ASV3R6 is set up for use by the cross-compiler to locate the header files, so as not to interfere with regular OS/2 native compilations that use the INCLUDE environment variable.

Programming

Once all the software is installed, then you can begin programming.

The following compilation takes the *module* source code from the OS/2 platform and produces object code in the target AS/400 *object-library*:

```
iccas /ASlobject-library /ASi- /C /J- /Lf /Ls /Q /Ti
  module.cpp
```

The following link-edit binds the AS/400 *module* object code into an executable *program* using the MQSeries C++ binding. The link-edit is performed remotely from the OS/2 platform using `ctthcmd`: the same command, without `ctthcmd`, can be executed natively on the AS/400 platform:

```
ctthcmd CRTPGM PGM(executable-library/program)
  MODULE(object-library/module)
  BNDSRVPGM(QMQM/IMQB23I4 QMQM/IMQS23I4)
  TEXT('Sample Program')
```

The following native AS/400 command executes a *program* from the *executable-library*. The MQSeries C++ sample executables can be found, along with the MQSeries C++ service programs, in the `QMQM` library:

```
CALL PGM(executable-library/program)
  PARM("parameter-1" "parameter-2")
```

The following command executes the HELLO WORLD sample program, which uses SYSTEM.DEFAULT.LOCAL.QUEUE:

```
CALL PGM(QMQM/IMQWRLDS)
```

Compiling VisualAge C++ sample programs for Windows 95 and NT

This section is aimed at the C++ programmer, who wishes to write VisualAge programs that will run on the Windows 95 and Windows NT platforms.

The IBM VisualAge for C++ for Windows run-time library `cppwm35i.dll` is used by MQSeries C++ and is redistributed, using the DLLRNAME utility from the VisualAge product, under the name `imqwm35i.dll`. Using DLLRNAME, you and your customers can also use the redistributed file, rather than supplying a redistribution copy of your own.

To use the MQSeries redistributed file, you need to process your executables after construction. Build your executable application in the normal way, whether it is a dynamic link library or a program, and then type:

```
dllrname applicname cppwm35i=imqwm35i
```

to rebind the application `applicname`.

Appendix B. MQI cross-reference

Read this information together with the *MQSeries Application Programming Reference*.

Table 4. Data structure, class, and file cross-reference

Data Structure	Class	Include file
	ImqBinary	imqbin.hpp
	ImqCache	imqcac.hpp
MQDLH	ImqDeadLetterHeader	imqdlh.hpp
MQOR	ImqDistributionList	imqdst.hpp
	ImqError	imqerr.hpp
MQGMO	ImqGetMessageOptions	imqgmo.hpp
	ImqHeader	imqhdr.hpp
MQIIH	ImqImsBridgeHeader	imqiih.hpp
	ImqItem	imqitm.hpp
MQMD	ImqMessage	imqmsg.hpp
	ImqMessageTracker	imqmtr.hpp
MQOD, MQRR	ImqObject	imqobj.hpp
MQPMO, MQPMR, MQRR	ImqPutMessageOptions	imqpmo.hpp
	ImqProcess	imqpro.hpp
	ImqQueue	imqque.hpp
MQBO, MQCNO	ImqQueueManager	imqmgr.hpp
MQRMH	ImqReferenceHeader	imqrfh.hpp
	ImqString	imqstr.hpp
MQTM	ImqTrigger	imqtrg.hpp
MQTMC		
MQTMC2	ImqTrigger	imqtrg.hpp
MQXQH		

Table 5 (Page 1 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqCache	**automatic buffer**			MQGET
ImqCache	**buffer length**			MQGET
ImqCache	**buffer pointer**			MQGET, MQPUT
ImqCache	**data length**			MQGET
ImqCache	**data offset**			MQGET

Table 5 (Page 2 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqCache	**data pointer**			MQGET
ImqCache	**message length**			MQGET, MQPUT
ImqDeadLetterHeader	**dead-letter reason code**	MQDLH	Reason	
ImqDeadLetterHeader	**destination queue manager name**	MQDLH	DestQMgrName	
ImqDeadLetterHeader	**destination queue name**	MQDLH	DestQName	
ImqDeadLetterHeader	**put application name**	MQDLH	PutApplName	
ImqDeadLetterHeader	**put application type**	MQDLH	PutApplType	
ImqDeadLetterHeader	**put date**	MQDLH	PutDate	
ImqDeadLetterHeader	**put time**	MQDLH	PutTime	
ImqError	**completion code**			MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET
ImqError	**reason code**			MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET
ImqGetMessageOptions	**group status**	MQGMO	GroupStatus	
ImqGetMessageOptions	**match options**	MQGMO	MatchOptions	
ImqGetMessageOptions	**options**	MQGMO	Options	

Table 5 (Page 3 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqGetMessageOptions	**resolved queue name**	MQGMO	ResolvedQName	
ImqGetMessageOptions	**segmentation**	MQGMO	Segmentation	
ImqGetMessageOptions	**segment status**	MQGMO	SegmentStatus	
ImqGetMessageOptions		MQGMO	Signal1	
ImqGetMessageOptions		MQGMO	Signal2	
ImqGetMessageOptions	**sync-point participation**	MQGMO	Options	
ImqGetMessageOptions	**wait interval**	MQGMO	WaitInterval	
ImqHeader	**character set**	MQDLH, MQIIH	CodedCharSetId	
ImqHeader	**encoding**	MQDLH, MQIIH	Encoding	
ImqHeader	**format**	MQDLH, MQIIH	Format	
ImqHeader	**header flags**	MQIIH, MQRMH	Flags	
ImqImsBridgeHeader	**authenticator**	MQIIH	Authenticator	
ImqImsBridgeHeader	**commit mode**	MQIIH	CommitMode	
ImqImsBridgeHeader	**logical terminal override**	MQIIH	LTermOverride	
ImqImsBridgeHeader	**message format services map name**	MQIIH	MFSMapName	
ImqImsBridgeHeader	**reply-to format**	MQIIH	ReplyToFormat	
ImqImsBridgeHeader	**security scope**	MQIIH	SecurityScope	
ImqImsBridgeHeader	**transaction instance id**	MQIIH	TranInstanceId	
ImqImsBridgeHeader	**transaction state**	MQIIH	TranState	
ImqItem	**structure id**			MQGET
ImqMessage	**application id data**	MQMD	ApplIdentityData	
ImqMessage	**application origin data**	MQMD	ApplOriginData	

Table 5 (Page 4 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqMessage	**backout count**	MQMD	BackoutCount	
ImqMessage	**character set**	MQMD	CodedCharSetId	
ImqMessage	**encoding**	MQMD	Encoding	
ImqMessage	**expiry**	MQMD	Expiry	
ImqMessage	**format**	MQMD	Format	
ImqMessage	**message flags**	MQMD	MsgFlags	
ImqMessage	**message type**	MQMD	MsgType	
ImqMessage	**offset**	MQMD	Offset	
ImqMessage	**original length**	MQMD	OriginalLength	
ImqMessage	**persistence**	MQMD	Persistence	
ImqMessage	**priority**	MQMD	Priority	
ImqMessage	**put application name**	MQMD	PutApplName	
ImqMessage	**put application type**	MQMD	PutApplType	
ImqMessage	**put date**	MQMD	PutDate	
ImqMessage	**put time**	MQMD	PutTime	
ImqMessage	**reply-to queue manager name**	MQMD	ReplyToQMgr	
ImqMessage	**reply-to queue name**	MQMD	ReplyToQ	
ImqMessage	**report**	MQMD	Report	
ImqMessage	**sequence number**	MQMD	MsgSeqNumber	
ImqMessage	**total message length**		DataLength	MQGET
ImqMessage	**user id**	MQMD	UserIdentifier	
ImqMessageTracker	**accounting token**	MQMD	AccountingToken	
ImqMessageTracker	**correlation id**	MQMD	CorrelId	
ImqMessageTracker	**feedback**	MQMD	Feedback	
ImqMessageTracker	**group id**	MQMD	GroupId	

Table 5 (Page 5 of 10). Object attribute cross-reference

Object	**Attribute**	**Data Structure**	**Field/Inquiry**	**Call**
ImqMessageTracker	**message id**	MQMD	MsgId	
ImqObject	**alternate user id**	MQOD	AlternateUserId	
ImqObject	**close options**			MQCLOSE
ImqObject	**connection reference**			
ImqObject	**description**		MQCA_Q_DESC, MQCA_Q_MGR_DESC, MQCA_PROCESS_DESC	MQINQ
ImqObject	**name**	MQOD	ObjectName, MQCA_Q_MGR_NAME, MQCQ_Q_NAME, MQCA_PROCESS_NAME	MQINQ
ImqObject	**next managed object**			
ImqObject	**open options**			MQOPEN
ImqObject	**open status**			MQOPEN, MQCLOSE
ImqObject	**previous managed object**			
ImqProcess	**application type**		MQIA_APPL_TYPE	MQINQ
ImqProcess	**application id**		MQCA_APPL_ID	MQINQ
ImqProcess	**environment data**		MQCA_ENV_DATA	MQINQ
ImqProcess	**user data**		MQCA_USER_DATA	MQINQ
ImqPutMessageOptions	**context reference**	MQPMO	Context	
ImqPutMessageOptions		MQPMO	InvalidDestCount	
ImqPutMessageOptions		MQPMO	KnownDestCount	
ImqPutMessageOptions	**options**	MQPMO	Options	
ImqPutMessageOptions	**record fields**	MQPMO	PutMsgRecFields	
ImqPutMessageOptions	**resolved queue manager name**	MQPMO	ResolvedQMgrName	
ImqPutMessageOptions	**resolved queue name**	MQPMO	ResolvedQName	
ImqPutMessageOptions		MQPMO	Timeout	

Table 5 (Page 6 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqPutMessageOptions		MQPMO	UnknownDestCount	
ImqPutMessageOptions	**sync-point participation**	MQPMO	Options	
ImqQueue	**backout requeue name**		MQCA_BACKOUT_REQ_Q_NAME	MQINQ
ImqQueue	**backout threshold**		MQIA_BACKOUT_THRESHOLD	MQINQ
ImqQueue	**base queue name**		MQCA_BASE_Q_NAME	MQINQ
ImqQueue	**creation date**		MQCA_CREATION_DATE	MQINQ
ImqQueue	**creation time**		MQCA_CREATION_TIME	MQINQ
ImqQueue	**current depth**		MQIA_CURRENT_Q_DEPTH	MQINQ
ImqQueue	**default input open option**		MQIA_DEF_INPUT_OPEN_OPTION	MQINQ
ImqQueue	**default persistence**		MQIA_DEF_PERSISTENCE	MQINQ
ImqQueue	**default priority**		MQIA_DEF_PRIORITY	MQINQ
ImqQueue	**definition type**		MQIA_DEFINITION_TYPE	MQINQ
ImqQueue	**depth high event**		MQIA_Q_DEPTH_HIGH_EVENT	MQINQ
ImqQueue	**depth high limit**		MQIA_Q_DEPTH_HIGH_LIMIT	MQINQ
ImqQueue	**depth low event**		MQIA_Q_DEPTH_LOW_EVENT	MQINQ
ImqQueue	**depth low limit**		MQIA_Q_DEPTH_LOW_LIMIT	MQINQ
ImqQueue	**depth maximum event**		MQIA_Q_DEPTH_MAX_LIMIT	MQINQ
ImqQueue	**distribution list reference**			
ImqQueue	**distribution lists**		MQIA_DIST_LISTS	MQINQ, MQSET
ImqQueue	**dynamic queue name**	MQOD	DynamicQName	

Table 5 (Page 7 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqQueue	**harden get backout**		MQIA_HARDEN_GET_BACKOUT	MQINQ
ImqQueue	**inhibit get**		MQIA_INHIBIT_GET	MQINQ, MQSET
ImqQueue	**inhibit put**		MQIA_INHIBIT_PUT	MQINQ, MQSET
ImqQueue	**initiation queue name**		MQCA_INITIATION_Q_NAME	MQINQ
ImqQueue	**maximum depth**		MQIA_MAX_Q_DEPTH	MQINQ
ImqQueue	**maximum message length**		MQIA_MAX_MSG_LENGTH	MQINQ
ImqQueue	**message delivery sequence**		MQIA_MSG_DELIVERY_SEQUENCE	MQINQ
ImqQueue	**next distributed queue**			
ImqQueue	**open input count**		MQIA_OPEN_INPUT_COUNT	MQINQ
ImqQueue	**open output count**		MQIA_OPEN_OUTPUT_COUNT	MQINQ
ImqQueue	**previous distributed queue**			
ImqQueue	**process name**		MQCA_PROCESS_NAME	MQINQ
ImqQueue	**queue manager name**	MQOD	ObjectQMgrName	
ImqQueue	**queue type**		MQIA_Q_TYPE	MQINQ
ImqQueue	**remote queue manager name**		MQCA_REMOTE_Q_MGR_NAME	MQINQ
ImqQueue	**remote queue name**		MQCA_REMOTE_Q_NAME	MQINQ
ImqQueue	**retention interval**		MQIA_RETENTION_INTERVAL	MQINQ
ImqQueue	**scope**		MQIA_SCOPE	MQINQ
ImqQueue	**service interval**		MQIA_Q_SERVICE_INTERVAL	MQINQ

Table 5 (Page 8 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqQueue	**service interval event**		MQIA_Q_SERVICE_INTERVAL_EVENT	MQINQ
ImqQueue	**shareability**		MQIA_SHAREABILITY	MQINQ
ImqQueue	**transmission queue name**		MQCA_XMIT_Q_NAME	MQINQ
ImqQueue	**trigger control**		MQIA_TRIGGER_CONTROL	MQINQ, MQSET
ImqQueue	**trigger data**		MQCA_TRIGGER_DATA	MQINQ, MQSET
ImqQueue	**trigger depth**		MQIA_TRIGGER_DEPTH	MQINQ, MQSET
ImqQueue	**trigger message priority**		MQIA_TRIGGER_MSG_PRIORITY	MQINQ, MQSET
ImqQueue	**trigger type**		MQIA_TRIGGER_TYPE	MQINQ, MQSET
ImqQueue	**usage**		MQIA_USAGE	MQINQ
ImqQueueManager	**authority event**		MQIA_AUTHORITY_EVENT	MQINQ
ImqQueueManager	**begin options**	MQBO	Options	MQBEGIN
ImqQueueManager	**character set**		MQIA_CODED_CHAR_SET_ID	MQINQ
ImqQueueManager	**command input queue name**		MQCA_COMMAND_INPUT_Q_NAME	MQINQ
ImqQueueManager	**command level**		MQIA_COMMAND_LEVEL	MQINQ
ImqQueueManager	**connect options**	MQCNO	Options	MQCONNX
ImqQueueManager	**connection status**			MQCONNX, MQDISC
ImqQueueManager	**dead-letter queue name**		MQCA_DEAD_LETTER_Q_NAME	MQINQ
ImqQueueManager	**default transmission queue name**		MQCA_DEF_XMIT_Q_NAME	MQINQ
ImqQueueManager	**distribution lists**		MQIA_DIST_LISTS	MQINQ

Table 5 (Page 9 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqQueueManager	**first distributed queue**			
ImqQueueManager	**inhibit event**		MQIA_INHIBIT_EVENT	MQINQ
ImqQueueManager	**local event**		MQIA_LOCAL_EVENT	MQINQ
ImqQueueManager	**maximum handles**		MQIA_MAX_HANDLES	MQINQ
ImqQueueManager	**maximum message length**		MQIA_MAX_MSG_LENGTH	MQINQ
ImqQueueManager	**maximum priority**		MQIA_MAX_PRIORITY	MQINQ
ImqQueueManager	**maximum uncommitted messages**		MQIA_MAX_UNCOMMITTED_MSGS	MQINQ
ImqQueueManager	**performance event**		MQIA_PERFORMANCE_EVENT	MQINQ
ImqQueueManager	**platform**		MQIA_PLATFORM	MQINQ
ImqQueueManager	**remote event**		MQIA_REMOTE_EVENT	MQINQ
ImqQueueManager	**start-stop event**		MQIA_START_STOP_EVENT	MQINQ
ImqQueueManager	**sync-point availability**		MQIA_SYNCPOINT	MQINQ
ImqQueueManager	**trigger interval**		MQIA_TRIGGER_INTERVAL	MQINQ
ImqReferenceHeader	**destination environment**	MQRMH	DestEnvLength, DestEnvOffset	
ImqReferenceHeader	**destination name**	MQRMH	DestNameLength, DestNameOffset	
ImqReferenceHeader	**instance id**	MQRMH	ObjectInstanceId	
ImqReferenceHeader	**logical length**	MQRMH	DataLogicalLength	
ImqReferenceHeader	**logical offset**	MQRMH	DataLogicalOffset	
ImqReferenceHeader	**logical offset 2**	MQRMH	DataLogicalOffset2	
ImqReferenceHeader	**reference type**	MQRMH	ObjectType	
ImqReferenceHeader	**source environment**	MQRMH	SrcEnvLength, SrcEnvOffset	
ImqReferenceHeader	**source name**	MQRMH	SrcNameLength, SrcNameOffset	

Table 5 (Page 10 of 10). Object attribute cross-reference

Object	Attribute	Data Structure	Field/Inquiry	Call
ImqTrigger	**application id**	MQTM	ApplId	
ImqTrigger	**application type**	MQTM	ApplType	
ImqTrigger	**environment data**	MQTM	EnvData	
ImqTrigger	**process name**	MQTM	ProcessName	
ImqTrigger	**queue name**	MQTM	QName	
ImqTrigger	**trigger data**	MQTM	TriggerData	
ImqTrigger	**user data**	MQTM	UserData	

Appendix C. Reason codes

The following reason codes can occur in addition to those documented for the MQSeries MQI.

MQRC_REOPEN_EXCL_INPUT_ERROR (6100 or X'17D4')
: An open object does not have the correct ImqObject **open options** and requires one or more additional options. An implicit reopen (see "Reopen" on page 15) is required but closure has been prevented.

  Closure has been prevented because the queue is open for exclusive input and closure might result in the queue being accessed by another process or thread, before the queue is reopened by the process or thread that presently has access.

  Corrective action: Set the **open options** explicitly to cover all eventualities so that implicit reopening is not required.

MQRC_REOPEN_INQUIRE_ERROR (6101 or X'17D5')
: An open object does not have the correct ImqObject **open options** and requires one or more additional options. An implicit reopen (see "Reopen" on page 15) is required but closure has been prevented.

  Closure has been prevented because one or more characteristics of the object need to be checked dynamically prior to closure, and the **open options** do not already include MQOO_INQUIRE.

  Corrective action: Set the **open options** explicitly to include MQOO_INQUIRE.

MQRC_REOPEN_SAVED_CONTEXT_ERR (6102 or X'17D6')
: An open object does not have the correct ImqObject **open options** and requires one or more additional options. An implicit reopen (see "Reopen" on page 15) is required but closure has been prevented.

  Closure has been prevented because the queue is open with MQOO_SAVE_ALL_CONTEXT, and a destructive get has been performed previously. This has caused retained state information to be associated with the open queue and this information would be destroyed by closure.

  Corrective action: Set the **open options** explicitly to cover all eventualities so that implicit reopening is not required.

MQRC_REOPEN_TEMPORARY_Q_ERROR (6103 or X'17D7')
: An open object does not have the correct ImqObject **open options** and requires one or more additional options. An implicit reopen (see "Reopen" on page 15) is required but closure has been prevented.

  Closure has been prevented because the queue is a local queue of the definition type MQQDT_TEMPORARY_DYNAMIC, that would be destroyed by closure.

  Corrective action: Set the **open options** explicitly to cover all eventualities so that implicit reopening is not required.

MQRC_ATTRIBUTE_LOCKED (6104 or X'17D8')
: An attempt has been made to change the value of an attribute of an object while that object is open, or, for an ImqQueueManager object, while that object is connected. Certain attributes cannot be changed in these

circumstances. Close or disconnect the object (as appropriate) before changing the attribute value.

An object may have been connected and/or opened unexpectedly and implicitly in order to perform an MQINQ call. Check the attribute cross-reference table (see Appendix B, “MQI cross-reference” on page 95) to determine whether any of your method invocations result in an MQINQ call.

Corrective action: Include MQOO_INQUIRE in the ImqObject **open options** and set them earlier.

MQRC_CURSOR_NOT_VALID (6105 or X'17D9')
: The browse cursor for an open queue has been invalidated since it was last used by an implicit reopen (see “Reopen” on page 15).

  Corrective action: Set the ImqObject **open options** explicitly to cover all eventualities so that implicit reopening is not required.

MQRC_ENCODING_ERROR (6106 or X'17DA')
: The encoding of the (next) message item needs to be MQENC_NATIVE for pasting.

MQRC_STRUC_ID_ERROR (6107 or X'17DB')
: The structure id for the (next) message item, which is derived from the 4 characters beginning at the data pointer, is either missing or is inconsistent with the class of object into which the item is being pasted.

MQRC_NULL_POINTER (6108 or X'17DC')
: A null pointer has been supplied where a non-null pointer is either required or implied.

MQRC_NO_CONNECTION_REFERENCE (6109 or X'17DD')
: The **connection reference** is null. A connection to an ImqQueueManager object is required.

MQRC_NO_BUFFER (6110 or X'17DE')
: No buffer is available. For an ImqCache object, one cannot be allocated, denoting an internal inconsistency in the object state that should not occur.

MQRC_BINARY_DATA_LENGTH_ERROR (6111 or X'17DF')
: The length of the binary data is inconsistent with the length of the target attribute. Zero is a correct length for all attributes. MQ_ACCOUNTING_TOKEN_LENGTH is the correct length for an **accounting token**. MQ_CORREL_ID_LENGTH is the correct length for a **correlation id**. MQ_GROUP_ID_LENGTH is the correct length for a **group id**. MQ_MSG_ID_LENGTH is the correct length for a **message id**. MQ_OBJECT_INSTANCE_ID_LENGTH is the correct length for an **instance id**. MQ_TRAN_INSTANCE_ID_LENGTH is the correct length for a **transaction instance id**.

MQRC_BUFFER_NOT_AUTOMATIC (6112 or X'17E0')
: A user-defined (and managed) buffer cannot be resized. A user-defined buffer can only be replaced or withdrawn. A buffer must be automatic (system-managed) before it can be resized.

MQRC_INSUFFICIENT_BUFFER (6113 or X'17E1')
: There is insufficient buffer space available after the data pointer to accommodate the request. This might be because the buffer cannot be resized.

MQRC_INSUFFICIENT_DATA (6114 or X'17E2')
: There is insufficient data after the data pointer to accommodate the request.

MQRC_DATA_TRUNCATED (6115 or X'17E3')
: Data has been truncated when copying from one buffer to another. This might be because the target buffer cannot be resized, or because there is a problem addressing one or other buffer, or because a buffer is being downsized with a smaller replacement.

MQRC_ZERO_LENGTH (6116 or X'17E4')
: A zero length has been supplied where a positive length is either required or implied.

MQRC_INCONSISTENT_FORMAT (6119 or X'17E7')
: The format of the (next) message item is inconsistent with the class of object into which the item is being pasted.

MQRC_INCONSISTENT_OBJECT_STATE (6120 or X'17E8')
: There is an inconsistency between this object, which is open, and the referenced ImqQueueManager object, which is not connected.

MQRC_CONTEXT_OBJECT_NOT_VALID (6121 or X'17E9')
: The ImqPutMessageOptions **context reference** does not reference a valid ImqQueue object. The object has been previously destroyed.

MQRC_CONTEXT_OPEN_ERROR (6122 or X'17EA')
: The ImqPutMessageOptions **context reference** references an ImqQueue object that could not be opened to establish a context. This may be because the ImqQueue object has inappropriate **open options**. Inspect the referenced object **reason code** to establish the cause.

MQRC_STRUC_LENGTH_ERROR (6123 or X'17EB')
: The length of a data structure is inconsistent with its content. For an MQRMH, the length is insufficient to contain the fixed fields and all offset data.

Appendix D. Notices

Programming interface information

This book is intended to help you to write application programs that run under MQSeries C++. This book documents General-use Programming Interface and Associated Guidance Information provided by MQSeries C++.

General-use programming interfaces allow the customer to write program that obtain the services of MQSeries.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	AS/400	BookManager
Client Access	Client Access/400	CICS
FFST	First Failure Support Technology	IBM
IMS	MQ	MQSeries
MQSeries Three Tier	MVS/ESA	OS/2
OS/400	RACF	VisualAge
VSE/ESA		

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary of terms and abbreviations

This glossary defines MQSeries terms and abbreviations used in this book. If you do not find the term you are looking for, see the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

abstract class. A class that can only be instantiated as a derivation.

attribute. A property of an object or class, which can be distinguished distinctly from any other properties. Attributes often describe state information.

B

behavior. The functionality embodied within a method.

C

class. An abstract model of behavior; a collection of methods. A class typically provides some unique behavior, in addition to other, common, behavior. The distinction between unique and common behavior is effected using either inheritance, or multiple interfaces.

class hierarchy. Classes related by inheritance.

class library. A bundled collection of classes, usually related.

constructor. A special method used to initialize an object.

D

derivation. The refinement or extension of one class from another.

E

encapsulation. The restriction whereby class behavior may only be observed using the methods of that class.

exclusive method. A method that is not intended to exhibit polymorphism; one with specific effect.

F

friend class. A class that is regarded as being derived from another, while this is not the case, for the purpose of accessing protected methods and instance data.

function. A classic function call such as is supported by the C programming language.

I

inheritance. The ability of a class to include the behavior of another through refinement and extension; only refined and extended methods are defined in the derived class, thereby preserving encapsulation.

instance. An object.

instance data. State information associated with an object.

interface. An abstract model of behavior; a collection of functions or methods.

M

marshalling. The serialization of data.

method. A means of invoking a particular behavior in an object or class.

O

object. In C an object is an instance of a class.

overloading. The existence of more than one flavor of method with the same name or operator, but with different signatures, within a class; while the name or operator remains the same, the method parameters differ, each signature requiring a separate implementation. Such methods usually exhibit the same behavior, despite differences in signature.

P

parent class. A class from which another is derived.

polymorphism. The characteristic whereby a method can be applied to a variety of classes, with consequent various effects: for example, an “open” method could be applied equally to “book” and “door” class objects.

private methods and instance data. Methods and instance data that are only accessible to the implementation of the same class.

protected methods and instance data. Methods and instance data that are only accessible to the implementations of the same or derived classes, or from friend classes.

public methods and instance data. Methods and instance data that are accessible to all classes.

S

serialization. The writing of data in sequential fashion to a communications medium from program memory.

signature. A distinct combination of method name or operator, and parameters.

streaming. The marshalling of class information and object instance data.

T

this. The reserved word that represents a pointer to the current object.

type. A fundamental data type of computer architecture, including for example character string and integer.

V

virtual method. A method that exhibits polymorphism.

Index

Sending your comments to IBM

MQSeries

Using C++

SC33-1877-01

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 request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should 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, use the Readers' Comment Form
- By fax:
  - From outside the U.K., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: WINVMD(IDRCF)
  - Internet: idrcf@winvmd.vnet.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries

Using C++

SC33-1877-01

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Telephone	Email

MQSeries

MQSeries using C++   SC33-1877-01

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States of America
Belgium	France	Israel	New Zealand	Spain	
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ   United Kingdom

3 Fold along this line

From: Name ______________________
Company or Organization ______________________
Address ______________________
____
EMAIL ______________________
Telephone ______________________

1 Cut along this line

4 Fasten here with adhesive tape

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-1877-01