MQSeries®

Using C++

SC33-1877-03

MQSeries®

Using C++

SC33-1877-03

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix D. Notices" on page 137.

Fourth edition (March 2000)

This edition applies to the following products:

- MQSeries for AIX® Version 5.1
- MQSeries for AS/400® Version 5.1
- MQSeries for HP-UX Version 5.1
- MQSeries for OS/2® Warp Version 5.1
- MQSeries for OS/390® Version 2.1
- MQSeries for Sun Solaris Version 5.1
- MQSeries for Windows NT® Version 5.1

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

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 and later
- MQSeries for AS/400 Version 4 Release 2 and later
- MQSeries for HP-UX Version 5 and later
- MQSeries for OS/2 Warp Version 5 and later
- MQSeries for OS/390 Version 2.1 and later
- MQSeries for Sun Solaris Version 5 and later
- MQSeries for Windows NT Version 5 and later

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

What you need to know to understand this book

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

The base directory

Throughout this book, the name **mqmtop** has been used to represent the name of the base directory where MQSeries is typically installed.
- For MQSeries for OS/2 Warp and MQSeries for Windows NT, **mqmtop** represents the directory **\mqm**. On MQSeries for Windows NT, it may sometimes represent the directory **\Program Files\MQSeries**.
- For MQSeries for AIX, **mqmtop** represents the directory **/usr/mqm**.
- For other UNIX systems, the name of the actual directory is **/opt/mqm**.

How to use this book

First read "Chapter 1. Introduction to MQSeries C++" on page 1. This chapter is a guide to programming in C++ for MQSeries, 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 21.

“Chapter 3. MQSeries C++ classes” on page 25. is the main, reference part of the book. This should be read together with “Appendix B. MQI cross-reference” on page 121 and the *MQSeries Application Programming Reference*.

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

There is a glossary and a bibliography at the back of the book.

Summary of changes

This information describes changes to the MQSeries product and changes to this edition of the *MQSeries Using C++* book. Changes since the previous edition of the book are marked in the left-hand margin with vertical bars

Changes for this edition (SC33-1877-03)

For additional new function in MQSeries for AS/400 V5.1, see “Compiling C++ sample programs for AS/400” on page 118.

Changes for the third edition (SC33-1877-02)

- MQSeries for C++ is now supplied as part of MQSeries for OS/390 V2.1.
- Three new MQSeries C++ classes are supported:
  - ImqCicsBridgeHeader
  - ImqNameList
  - ImqWorkHeader

Changes for the second edition (SC33-1877-01)

MQSeries C++ is 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.

Changes

Chapter 1. Introduction to MQSeries C++

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

This chapter describes the following:

- "Features of MQSeries C++"
- "Preparing message data" on page 4
- "Reading messages" on page 5
- "Writing a message to the dead-letter queue" on page 11
- "Writing a message to the IMS bridge" on page 12
- "Writing a message to the CICS bridge" on page 13
- "Writing a message to the work header" on page 14
- "Sample programs" on page 14
- "Implicit operations" on page 18
- "Binary and character strings" on page 19
- "Unsupported functions" on page 20

MQSeries C++ can be used with the following products when they have been installed as a full queue manager:

- MQSeries for AIX Version 5 and later
- MQSeries for AS/400 Version 4 Release 2 and later
- MQSeries for HP-UX Version 5 and later
- MQSeries for OS/2 Warp Version 5 and later
- MQSeries for OS/390
- MQSeries for Sun Solaris Version 5 and later
- MQSeries for Windows NT Version 5 and later

MQSeries C++ can also be used with an MQSeries client supplied with the 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
- CICS® Bridge header transmission and receipt
- Work header transmission and receipt

The following Booch class diagrams show that all the classes are broadly parallel to those MQSeries entities in the procedural MQI (for example C) that have either handles or data structures. All classes inherit from the ImqError class (see "ImqError" on page 44), which allows an error condition to be associated with each object.

To interpret Booch class diagrams correctly, you must be aware of the following:

- Methods and noteworthy attributes are listed below the *class* name.
- A small triangle within a cloud denotes an *abstract class*.
- *Inheritance* is denoted by an arrow to the parent class.
- An undecorated line between clouds denotes a *cooperative relationship* between classes.
- A line decorated with a number denotes a *referential relationship* between two classes. The number indicates the number of objects that may participate in a given relationship at any one time.

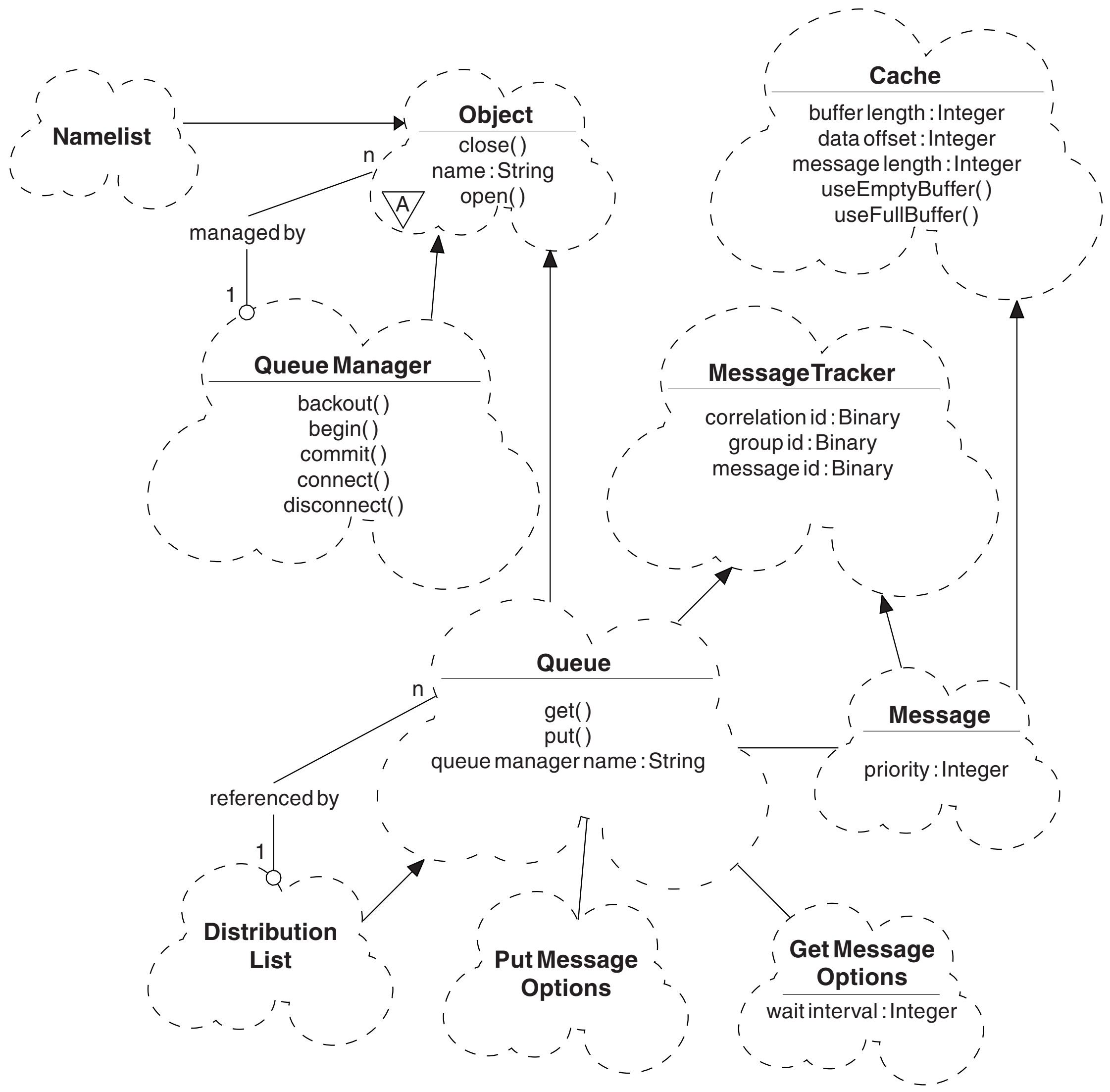


Figure 1. MQSeries C++ classes (queue management)

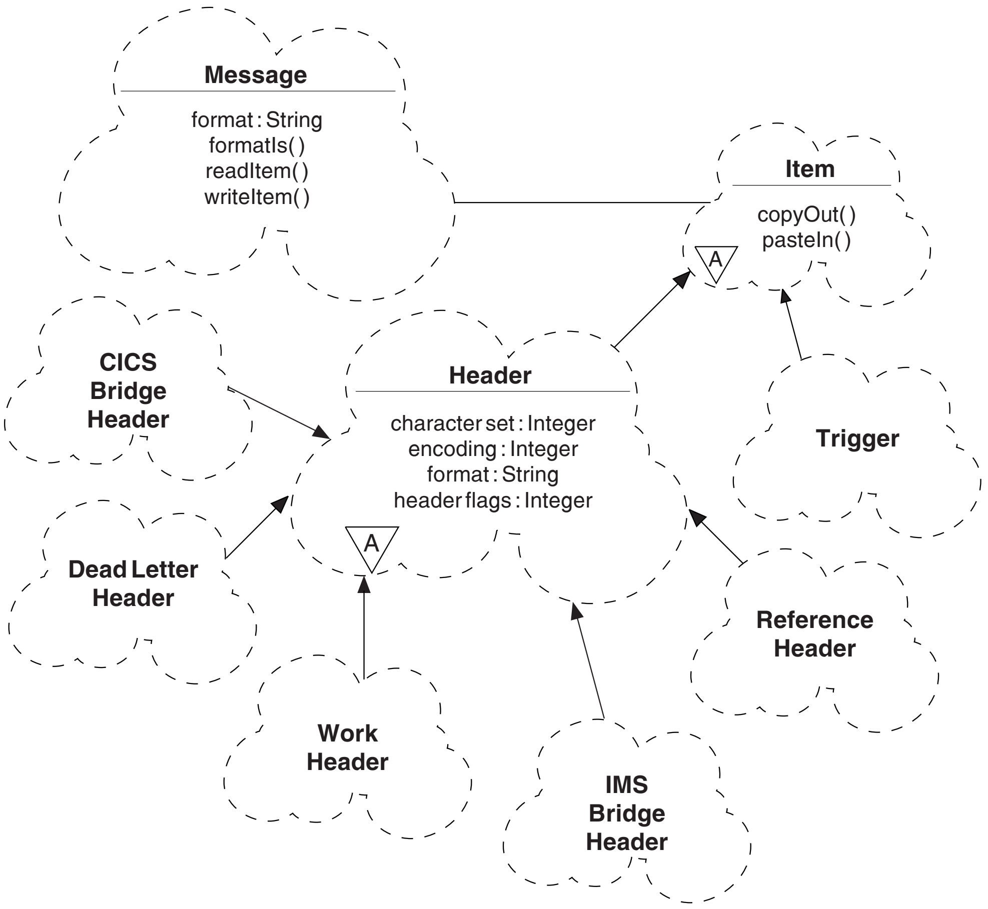


Figure 2. MQSeries C++ classes (item handling)

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 27), 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 105), 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 121) are accessed with methods.

Entities with handles come under the ImqObject class hierarchy (see “ImqObject” on page 70) 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 57) 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 29). A buffer is associated (by inheritance) with each ImqMessage object (see “ImqMessage” on page 57): it can be supplied by the application (using either the **useEmptyBuffer** or **useFullBuffer** method) or 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, 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 multipart message can be written in sequential pieces.

Figure 3 on page 5 shows 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.

Reading messages

Figure 4 shows a message receipt that takes account of a number of 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.                             */
        ...
      }
      /* 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. */
    }
```

Figure 4. Retrieving items within a message (Part 1 of 2)

```
    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.  */

      /* 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. Retrieving items within a message (Part 2 of 2)

In Figure 4, `FMT_USERCLASS` is a constant representing the 8-character format name associated with an object of class `UserClass`, and is defined by the application.

`UserClass` would be derived from the ImqItem class (see “ImqItem” on page 55), and would implement the virtual **copyOut** and **pasteIn** methods from that class.

Reading messages

Figure 5 and Figure 6 on page 9 show example code from the ImqDeadLetterHeader class (see “ImqDeadLetterHeader” on page 39).

```
// Insert a dead-letter header.
// Return TRUE if successful.
ImqBoolean ImqDeadLetterHeader :: copyOut ( ImqMessage & msg ) {
  ImqBoolean bSuccess ;
  if ( msg.moreBytes( sizeof( omqdlh ) ) ) {
    ImqCache cacheData( msg ); // Preserve the original message content.
    // Note the original message attributes in the dead-letter header.
    setEncoding( msg.encoding( ) );
    setCharacterSet( msg.characterSet( ) );
    setFormat(msg.format( ) );

    // Set the message attributes to reflect the dead-letter header.
    msg.setEncoding( MQENC_NATIVE );
    msg.setCharacterSet( MQCCSI_Q_MGR );
    msg.setFormat( MQFMT_DEAD_LETTER_HEADER );
    // Replace the existing data with the dead-letter header.
    msg.clearMessage( );
    if ( msg.write( sizeof( omqdlh ), (char *) & omqdlh ) ) {
      // Append the original message data.
      bSuccess = msg.write( cacheData.messageLength( ),
                            cacheData.bufferPointer( ) );
    } else {
      bSuccess = FALSE ;
    }
  } else {
    bSuccess = FALSE ;
  }
  // Reflect and cache error in this object.
  if ( ! bSuccess ) {
    setReasonCode( msg.reasonCode( ) );
    setCompletionCode( msg.completionCode( ) );
  }

  return bSuccess ;
}
```

Figure 5. Custom encapsulated message-writing code

```
// Read a dead-letter header.
// Return TRUE if successful.
ImqBoolean ImqDeadLetterHeader :: pasteIn ( ImqMessage & msg ) {
  ImqBoolean bSuccess = FALSE ;

  // First check that the eye-catcher is correct.
  // This is also our guarantee that the "character set" is correct.
  if ( ImqItem::structureIdIs( MQDLH_STRUC_ID, msg ) ) {
    // Next check that the "encoding" is correct, as the MQDLH contains
    // numeric data.
    if ( msg.encoding( ) == MQENC_NATIVE ) {

      // Finally check that the "format" is correct.
      if ( msg.formatIs( MQFMT_DEAD_LETTER_HEADER ) ) {
        char * pszBuffer = (char *) & omqdlh ;
        // Transfer the MQDLH from the message and move the pointer on.
        if ( bSuccess = msg.read( sizeof( omdlh ), pszBuffer ) ) {
          // Update the encoding, character set and format of the message
          // to reflect the remaining data.
          msg.setEncoding( encoding( ) );
          msg.setCharacterSet( characterSet( ) );
          msg.setFormat( format( ) );
        } else {

          // Reflect the cache error in this object.
          setReasonCode( msg.reasonCode( ) );
          setCompletionCode( msg.completeionCode( ) );
        }
      } else {
        setReasonCode( MQRC_INCONSISTENT_FORMAT );
        setCompletionCode( MQCC_FAILED );
      }
    } else {
      setReasonCode( MQRC_ENCODING_ERROR );
      setCompletionCode( MQCC_FAILED );
    {
  } else {
    setReasonCode( MQRC_STRUC_ID_ERROR );
    setCompletionCode( MQCC_FAILED );
  }

  return bSuccess ;
}
```

Figure 6. Custom encapsulated message-reading code

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, nonautomatic 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 on the design of the application.

Reading messages

Figure 7 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 7. Retrieving messages into a fixed area of storage

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

In the fragment shown in Figure 7, the buffer can always be addressed directly, with *pszBuffer*, as opposed to using the **bufferPointer** method, although it is recommended 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 multipart 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 8. Writing a message to the dead-letter queue

Writing a message to the IMS bridge

Messages sent to the MQSeries-IMS bridge may use a special header. The IMS bridge header is prefixed to regular message data.

```
ImqQueueManager    mgr;         // The queue manager.
ImqQueue           queueBridge; // IMS bridge message queue.
ImqMessage         msg;         // Outgoing message.
ImqIMSBridgeHeader header;      // IMS Bridge header.

// Set up the message.
//
// Here we are constructing a message with format MQFMT_IMS_VAR_STRING,
// and appropriate data.
//
msg.write( 2,      /* ? */ );         // Total message length.
msg.write( 2,      /* ? */ );         // IMS flags.
msg.write( 7,      /* ? */ );         // Transaction code.
msg.write( /* ? */, /* ? */ );        // String data.
msg.setFormat( MQFMT_IMS_VAR_STRING ); // The format attribute.

// 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 into the message.
//
// This will:
// 1) Insert the header into the message buffer, before the existing
//    data.
// 2) Copy attributes out of the message descriptor into the header,
//    for example the IMS bridge header format attribute will now
//    be set to MQFMT_IMS_VAR_STRING.
// 3) Set up the message attributes to describe the header, in
//    particular setting the message format to MQFMT_IMS.
//
msg.writeItem( header );

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

Figure 9. Writing a message to the IMS bridge

Writing a message to the CICS bridge

Messages sent to MQSeries for OS/390 via the CICS bridge require a special header. The CICS bridge header is prefixed to regular message data.

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

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

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

// Insert the CICS 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 CICS bridge queue.
queueBridge.setConnectionReference( mgr );
queueBridge.setName( /* ? */ );
queueBridge.put( msg );
```

Figure 10. Writing a message to the CICS bridge

Writing a message to the work header

Messages sent to MQSeries for OS/390, which are destined for a queue managed by the OS/390 Workload Manager, require a special header. The work header is prefixed to regular message data.

```
ImqQueueManager mgr ;            // The queue manager.
ImqQueue queueIn ;               // Incoming message queue.
ImqQueue queueWLM ;              // WLM managed queue.
ImqMessage msg ;                 // Incoming and outgoing message.
ImqWorkHeader header ;           // Work header information

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

// Insert the Work 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 WLM managed queue.
queueWLM.setConnectionReference( mgr );
queueWLM.setName( /* ? */ );
queueWLM.put( msg );
```

Figure 11. Writing a message to the work header

Sample programs

The sample programs are:
- HELLO WORLD (imqwrld.cpp)
- SPUT (imqsput.cpp) and SGET (imqsget.cpp)
- DPUT (imqdput.cpp)

Note: DPUT is not supported on OS/390.

| The sample programs are located in the directories shown in Table 1.

Note:

The name **mqmtop** has been used to represent the name of the base directory where MQSeries is typically installed.
- For MQSeries for OS/2 Warp and MQSeries for Windows NT, **mqmtop** represents the directory **\mqm**. On MQSeries for Windows NT, it may sometimes represent the directory **\Program Files\MQSeries**.
- For MQSeries for AIX, **mqmtop** represents the directory **/usr/mqm**.
- For other UNIX systems, the name of the actual directory is **/opt/mqm**.

| *Table 1. Location of sample programs*

Environment	Directory containing source	Directory containing built programs
AIX	<**mqmtop**>/samp	<**mqmtop**>/samp/bin/ia
AS/400	/QIBM/ProdData/mqm/samp/	(see note 1)

Table 1. Location of sample programs (continued)

Environment	Directory containing source	Directory containing built programs
HP-UX	<**mqmtop**>/samp	<**mqmtop**>/samp/bin/ah and <**mqmtop**>/samp/bin/hh. (see note 2)
OS/2 Warp	<**mqmtop**>\tools\cplus\samples	<**mqmtop**>\tools\cplus\samples\bin\i2
OS/390	see page 119	see page 119
Sun Solaris	<**mqmtop**>/samp	<**mqmtop**>/samp/bin/ss
Windows NT, 95, and 98	<**mqmtop**>\tools\cplus\samples	<**mqmtop**>\tools\cplus\samples\bin\in and <**mqmtop**>\tools\cplus\samples\bin\vn (see note 3)
Windows 3.1 (16-bit client only)	bin\vw	bin\vw

Notes:

1. Programs built using the ILE C++ compiler for AS/400 are in the library QMQM. The include files are in /QIBM/ProdData/mqm/inc.
2. Programs built using the HP ANSII C++ compiler are found in directory <**mqmtop**>/samp/bin/ah. Programs built using the HP C++ Version 3.1 on HP-UX V10.20 and HP C++ Version 12.0 on HP-UX V11 are found in <**mqmtop**>/samp/bin/hh. For further information about these compilers, see “Compilers for MQSeries platforms” on page 117.
3. Programs built using the IBM VisualAge for C++ for Windows V3.5 compiler are found in directory <**mqmtop**>\tools\cplus\samples\bin\in. Programs built using the Microsoft™ Visual C++ V4.0 are found in <**mqmtop**>\tools\cplus\samples\bin\vn. For further information about these compilers, see “Compilers for MQSeries platforms” on page 117.

Sample program HELLO WORLD (imqwrld.cpp)

This program shows how to put and get a regular datagram (C structure) using the ImqMessage class. This sample, which is shown in Figure 12 on page 16, uses few method invocations, taking advantage of implicit method invocations such as **open**, **close**, and **disconnect**.

On all platforms except OS/390

If you are using a server connection to MQSeries:

1. Run[1] **imqwrlds** to use the existing default queue SYSTEM.DEFAULT.LOCAL.QUEUE.
2. Run **imqwrlds** SYSTEM.DEFAULT.MODEL.QUEUE to use a temporary dynamically assigned queue.

Note: If you are using a client connection to MQSeries, un **imqwrldc**.

On OS/390

- Construct and run a batch job, using the sample JCL **imqwrldr**. See “Running sample programs on OS/390” on page 120 for more information.

1. For details of executing C++ programs, see “Appendix A. Compiling and linking” on page 117.

```
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 ) {
    manager.setName( argv[ 2 ] );
  }
  if ( manager.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 );
    } 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 );
```

Figure 12. The HELLO WORLD sample program (Part 1 of 2)

```
      // 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( );
        // 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 12. The HELLO WORLD sample program (Part 2 of 2)

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

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

On all platforms except OS/390

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.

On OS/390

1. Construct and run a batch job using the sample JCL **imqsputr**. The messages are read from the SYSIN data set.
2. Construct and run a batch job using the sample JCL **imqsgetr**. The messages are retrieved from the queue and sent to the SYSPRINT data set.

See “Running sample programs on OS/390” on page 120 for more information.

These samples show the use of the following classes:

ImqError (see “ImqError” on page 44)
ImqMessage (see “ImqMessage” on page 57)
ImqObject (see “ImqObject” on page 70)
ImqQueue (see “ImqQueue” on page 82)
ImqQueueManager (see “ImqQueueManager” on page 94)

Sample program DPUT (imqdput.cpp)

This is a distribution list program that puts messages to a distribution list consisting of two queues. DPUT shows the use of the ImqDistributionList class (see “ImqDistributionList” on page 42). This sample is not supported on OS/390.

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.

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.[2]

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 121).

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.

2. Connect and open implicit behavior is controllable using class attributes.

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, 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 problems associated with the use of reopen.

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 105) 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 27) encapsulates binary byte arrays of arbitrary size. In particular it is used to hold these attributes:
accounting token (MQBYTE32)
correlation id (MQBYTE24)
facility token (MQBYTE8)

Binary and character strings

group id (MQBYTE24)
instance id (MQBYTE24)
message id (MQBYTE24)
message token (MQBYTE16)
transaction instance id (MQBYTE16)

of objects of these classes:
ImqCICSBridgeHeader (see "ImqCICSBridgeHeader" on page 32)
ImqGetMessageOptions (see "ImqGetMessageOptions" on page 46)
ImqIMSBridgeHeader (see "ImqIMSBridgeHeader" on page 52)
ImqMessageTracker (see "ImqMessageTracker" on page 64)
ImqReferenceHeader (see "ImqReferenceHeader" on page 102)
ImqWorkHeader (see "ImqWorkHeader" on page 114)

and provides support for comparison and copying.

Unsupported functions

The MQSeries C++ classes and methods are intended to be independent of MQSeries platform. They might therefore offer some functions that are not supported on certain platforms. If you attempt to use a function on a platform on which it is not supported, the function is detected by MQSeries but not by the C++ language bindings. MQSeries reports the error to your program, like any other MQI error.

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 2. 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 for 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 equal sign (=); the value after the equal sign indicates the default value that applies if the item is omitted.

All method names 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

The datatype **ImqBoolean** is represented by **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 13. 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 14. Declaration and use conventions

Notation

Chapter 3. MQSeries C++ classes

The MQSeries C++ classes encapsulate 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
Document links to other relevant classes, such as parent classes, and the classes of objects used in method signatures.

Object attributes
Attributes unique to the class. These 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 121). For detailed descriptions see the *MQSeries Application Programming Reference* book.

Constructors
Signatures of the special methods used to create an object of the class.

Object methods (public)
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)
Signatures of methods that do not require an instance of the class for their operation, and that have no usage restrictions.

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

Object methods (protected)
Signatures of methods that 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)
Implementation details for object instance data available to the implementations of derived classes. This section is of interest only to class writers, as opposed to class users.

Reason codes
MQRC_* values (see “Appendix C. Reason codes” on page 131) 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.

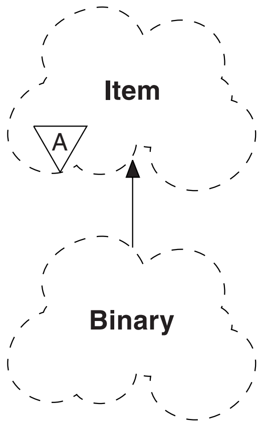
C++ classes

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 multithreaded 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.

The classes are:

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 55)

ImqMessage (see "ImqMessage" on page 57)

Object attributes

data An array of bytes of binary data. The initial value is null.

data length
The number of bytes. The initial value is zero.

data pointer
The address of the first byte of the **data**. The initial value is zero.

Constructors

ImqBinary( );
The default constructor.

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

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

Overloaded "ImqItem" methods

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

See the ImqItem class method description for further details.

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

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

See the ImqItem class method description for further details.

Object methods (public)

void operator = (const ImqBinary & *binary* **);**
Copies bytes from *binary*.

ImqBoolean operator == (const ImqBinary & *binary* **);**
Compares this object with *binary*. It returns FALSE if not equal and TRUE otherwise. The objects are equal if they have the same **data length** and the bytes match.

ImqBoolean copyOut(void * *buffer***, const size_t** *length***, const char** *pad* **= 0);**
Copies up to *length* bytes from the **data pointer** to *buffer*. If the **data length** is insufficient, the remaining space in *buffer* is filled with *pad* bytes. *buffer* may be zero if *length* is also zero. *length* must not be negative. It returns TRUE if successful.

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

ImqBoolean setDataLength(const size_t *length* **);**
Sets the **data length**. If the **data length** is changed as a result of this method, the data in the object is uninitialized. It returns TRUE if successful.

void * dataPointer( ) const ;
Returns the **data pointer**.

ImqBoolean isNull( ) const ;
Returns TRUE if the **data length** is zero, or if all of the **data** bytes are zero. Otherwise it returns FALSE.

ImqBoolean set(const void * *buffer***, const size_t** *length* **);**
Copies *length* bytes from *buffer*. It returns TRUE if successful.

Object methods (protected)

void clear( );
Reduces the **data length** to zero.

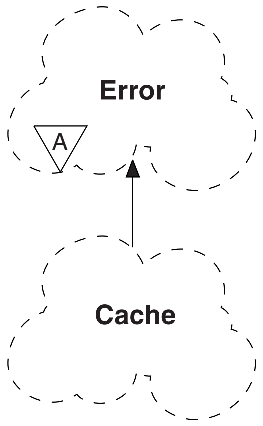
Reason codes

MQRC_NO_BUFFER

MQRC_STORAGE_NOT_AVAILABLE

MQRC_INCONSISTENT_FORMAT

ImqCache



| 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. This class relates to the MQI calls listed in Table 7 on page 122.

Other relevant classes

ImqError (see “ImqError” on page 44).

Object attributes

automatic buffer
Indicates whether buffer memory is managed automatically by the system (TRUE) or is supplied by the user (FALSE). This is initially set to TRUE.

Note: This attribute is not set directly, but is set indirectly using one of the **useEmptyBuffer** and **useFullBuffer** methods.

If user storage is supplied, this attribute is FALSE, buffer memory cannot grow, and buffer overflow errors may occur. The address and length of the buffer remain constant.

If user storage is not supplied, this attribute is TRUE, and buffer memory can grow incrementally to accommodate an arbitrary amount of message data. However, when the buffer grows, the address of the buffer may change, and so care has to be exercised when using the **buffer pointer** and **data pointer**.

buffer length
The number of bytes of memory in the buffer. The initial value is zero.

buffer pointer
The address of the buffer memory. The initial value is null.

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

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

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

message length
: The number of bytes of significant data in the buffer. The initial value is zero.

Constructors

ImqCache( );
: The default constructor.

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

Object methods (public)

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

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

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

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

void clearMessage( );
: Sets the **message length** and **data offset** 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**. This method returns TRUE if successful.

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

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

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

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

If **automatic buffer** is TRUE, more memory 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*. This method returns TRUE if successful.

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

Note: This method 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. This method returns TRUE if successful.

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

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

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

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

Reason codes

MQRC_BUFFER_NOT_AUTOMATIC

MQRC_DATA_TRUNCATED

MQRC_INSUFFICIENT_BUFFER

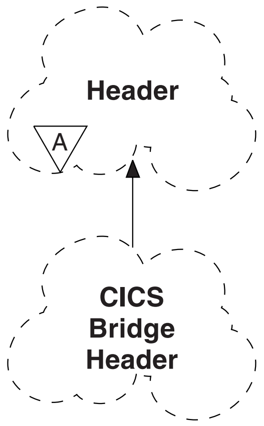
MQRC_INSUFFICIENT_DATA

MQRC_NULL_POINTER

MQRC_STORAGE_NOT_AVAILABLE

MQRC_ZERO_LENGTH

ImqCICSBridgeHeader



This class encapsulates specific features of the MQCIH data structure (see Table 8 on page 122). Objects of this class are used by applications that send messages to the CICS bridge through MQSeries for OS/390 V2.1.

Other relevant classes

ImqBinary (see "ImqBinary" on page 27)

ImqHeader (see "ImqHeader" on page 50)

ImqItem (see "ImqItem" on page 55)

ImqMessage (see "ImqMessage" on page 57)

ImqString (see "ImqString" on page 105)

Object attributes

ADS descriptor
Send/receive ADS descriptor. This is set using MQCADSD_NONE. The initial value is MQCADSD_NONE. The following additional values are possible:

MQCADSD_NONE

MQCADSD_SEND

MQCADSD_RECV

MQCADSD_MSGFORMAT

attention identifier
AID key. The field must be of length MQ_ATTENTION_ID_LENGTH.

authenticator
RACF® password or passticket. The initial value contains blanks, of length MQ_AUTHENTICATOR_LENGTH.

bridge abend code
Bridge abend code, of length MQ_ABEND_CODE_LENGTH. The initial value is four blank characters. The value returned in this field is dependent on the return code. See Table 3 on page 38 for more details.

bridge cancel code
Bridge abend transaction code. The field is reserved, must contain blanks, and be of length MQ_CANCEL_CODE_LENGTH.

bridge completion code
: Completion code, which can contain either the MQSeries completion code or the CICS EIBRESP value. The field has the initial value of MQCC_OK. The value returned in this field is dependent on the return code. See Table 3 on page 38 for more details.

bridge error offset
: Bridge error offset. The initial value is zero. This attribute is read-only.

bridge reason code
: Reason code. This field can contain either the MQSeries reason or the CICS EIBRESP2 value. The field has the initial value of MQRC_NONE. The value returned in this field is dependent on the return code. See Table 3 on page 38 for more details.

bridge return code
: Return code from the CICS bridge. The initial value is MQCRC_OK.

conversational task
: Indicates whether the task can be conversational. The initial value is MQCCT_NO. The following additional values are possible:

  MQCCT_YES

  MQCCT_NO

cursor position
: Cursor position. The initial value is zero.

facility keep time
: CICS bridge facility release time.

facility like
: Terminal emulated attribute. The field must be of length MQ_FACILITY_LIKE_LENGTH.

facility token
: BVT token value. The field must be of length MQ_FACILITY_LENGTH. The initial value is MQCFAC_NONE.

function
: Function, which can contain either the MQSeries call name or the CICS EIBFN function. The field has the initial value of MQCFUNC_NONE, with length MQ_FUNCTION_LENGTH. The value returned in this field is dependent on the return code. See Table 3 on page 38 for more details.

  The following additional values are possible when **function** contains an MQSeries call name:

  MQCFUNC_MQCONN

  MQCFUNC_MQGET

  MQCFUNC_MQINQ

  MQCFUNC_NONE

  MQCFUNC_MQOPEN

  MQCFUNC_PUT

  MQCFUNC_MQPUT1

get wait interval
: Wait interval for an MQGET call issued by the CICS bridge task. The field has an initial value of MQCGWI_DEFAULT. The field is applicable only when **uow control** has the value MQCUOWC_FIRST. The following additional values are possible:

MQCGWI_DEFAULT

MQWI_UNLIMITED

link type
Link type. The initial value is MQCLT_PROGRAM. The following additional values are possible:

MQCLT_PROGRAM

MQCLT_TRANSACTION

next transaction identifier
ID of the next transaction to attach. The field must be of length MQ_TRANSACTION_ID_LENGTH.

output data length
COMMAREA data length. The initial value is MQCODL_AS_INPUT.

reply-to format
Format name of the reply message. The initial value is MQFMT_NONE with length MQ_FORMAT_LENGTH.

start code
Transaction start code. The field must be of length MQ_START_CODE_LENGTH. The initial value is MQCSC_NONE. The following additional values are possible:

MQCSC_START

MQCSC_STARTDATA

MQCSC_TERMINPUT

MQCSC_NONE

task end status
Task end status. The initial value is MQCTES_NOSYNC. The following additional values are possible:

MQCTES_COMMIT

MQCTES_BACKOUT

MQCTES_ENDTASK

MQCTES_NOSYNC

transaction identifier
ID of the transaction to attach. The initial value must contain blanks, and must be of length MQ_TRANSACTION_ID_LENGTH. The field is applicable only when **uow control** has the value MQCUOWC_FIRST or MQCUOWC_ONLY.

UOW control
UOW control. The initial value is MQCUOWC_ONLY. The following additional values are possible:

MQCUOWC_FIRST

MQCUOWC_MIDDLE

MQCUOWC_LAST

MQCUOWC_ONLY

MQCUOWC_COMMIT

MQCUOWC_BACKOUT

MQCUOWC_CONTINUE

version
This is the MQCIH version number. The initial value is MQCIH_VERSION_2. The only other supported value is MQCIH_VERSION_1.

Constructors

ImqCICSBridgeHeader( );
The default constructor.

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

Overloaded "ImqItem" methods

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

See the parent class method description for more details.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **);**
Reads an MQCIH data structure form 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_CICS.

See the parent class method description for more details.

Object methods (public)

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

MQLONG ADSDescriptor( ) const;
Returns a copy of the **ADS descriptor**.

void setADSDescriptor(const MQLONG *descriptor* **= MQCADSD_NONE);**
Sets the **ADS descriptor**.

ImqString attentionIdentifier( ) const;
Returns a copy of the **attention identifier**, padded with trailing blanks to length MQ_ATTENTION_ID_LENGTH.

void setAttentionIdentifier(const char * *data* **= 0);**
Sets the **attention identifier**, padded with trailing blanks to length MQ_ATTENTION_ID_LENGTH. If no *data* is supplied, **attention identifier** is reset to the initial value.

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

void setAuthenticator(const char * *data* **= 0);**
Sets the **authenticator**, padded with trailing blanks to length MQ_AUTHENTICATOR_LENGTH. If no *data* is supplied, **authenticator** is reset to the initial value.

ImqString bridgeAbendCode( ) const;
: Returns a copy of the **bridge abend code**, padded with trailing blanks to length MQ_ABEND_CODE_LENGTH.

ImqString bridgeCancelCode( ) const;
: Returns a copy of the **bridge cancel code**, padded with trailing blanks to length MQ_CANCEL_CODE_LENGTH.

void setBridgeCancelCode(const char * *data* **= 0);**
: Sets the **bridge cancel code**, padded with trailing blanks to length MQ_CANCEL_CODE_LENGTH. If no *data* is supplied, the **bridge cancel code** is reset to the initial value.

MQLONG bridgeCompletionCode( ) const;
: Returns a copy of the **bridge completion code**.

MQLONG bridgeErrorOffset( ) const ;
: Returns a copy of the **bridge error offset**.

MQLONG bridgeReasonCode( ) const;
: Returns a copy of the **bridge reason code**.

MQLONG bridgeReturnCode( ) const;
: Returns the **bridge return code**.

MQLONG conversationalTask( ) const;
: Returns a copy of the **conversational task**.

void setConversationalTask(const MQLONG *task* **= MQCCT_NO);**
: Sets the **conversational task**.

MQLONG cursorPosition( ) const ;
: Returns a copy of the **cursor position**.

void setCursorPosition(const MQLONG *position* **= 0);**
: Sets the **cursor position**.

MQLONG facilityKeepTime( ) const;
: Returns a copy of the **facility keep time**.

void setFacilityKeepTime(const MQLONG *time* **= 0);**
: Sets the **facility keep time**.

ImqString facilityLike( ) const;
: Returns a copy of the **facility like**, padded with trailing blanks to length MQ_FACILITY_LIKE_LENGTH.

void setFacilityLike(const char * *name* **= 0);**
: Sets the **facility like**, padded with trailing blanks to length MQ_FACILITY_LIKE_LENGTH. If no *name* is supplied, **facility like** is reset to the initial value.

ImqBinary facilityToken( ) const;
: Returns a copy of the **facility token**.

ImqBoolean setFacilityToken(const ImqBinary & *token* **);**
: Sets the **facility token**. The **data length** of *token* must be either zero or MQ_FACILITY_LENGTH. It returns TRUE if successful.

void setFacilityToken(const MQBYTE8 *token* **= 0);**
: Sets the **facility token**. *token* may be zero, which is the same as specifying MQCFAC_NONE. If *token* is nonzero it must address MQ_FACILITY_LENGTH bytes of binary data. When using predefined

values such as MQCFAC_NONE, it may be necessary to make a cast to ensure a signature match. For example, (MQBYTE *)MQCFAC_NONE.

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

MQLONG getWaitInterval( ) const;
Returns a copy of the **get wait interval**.

void setGetWaitInterval(const MQLONG *interval* **= MQCGWI_DEFA**
Sets the **get wait interval**.

MQLONG linkType( ) const;
Returns a copy of the **link type**.

void setLinkType(const MQLONG *type* **= MQCLT_PROGRAM);**
Sets the **link type**.

ImqString nextTransactionIdentifier( ) const ;
Returns a copy of the **next transaction identifier** data, padded with trailing blanks to length MQ_TRANSACTION_ID_LENGTH.

MQLONG outputDataLength( ) const;
Returns a copy of the **output data length**.

void setOutputDataLength(const MQLONG *length* **= MQCODL_AS_INPUT);**
Sets the **output data length**.

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

void setReplyToFormat(const char * *name* **= 0);**
Sets the **reply-to format**, padded with trailing blanks to length MQ_FORMAT_LENGTH. If no *name* is supplied, **reply-to format** is reset to the initial value.

ImqString startCode( ) const;
Returns a copy of the **start code**, padded with trailing blanks to length MQ_START_CODE_LENGTH.

void setStartCode(const char * *data* **= 0);**
Sets the **start code** data, padded with trailing blanks to length MQ_START_CODE_LENGTH. If no *data* is supplied, **start code** is reset to the initial value.

MQLONG taskEndStatus( ) const;
Returns a copy of the **task end status**.

ImqString transactionIdentifier( ) const;
Returns a copy of the **transaction identifier** data, padded with trailing blanks to the length MQ_TRANSACTION_ID_LENGTH.

void setTransactionIdentifier(const char * *data* **= 0);**
Sets the **transaction identifier**, padded with trailing blanks to length MQ_TRANSACTION_ID_LENGTH. If no *data* is supplied, **transaction identifier** is reset to the initial value.

MQLONG UOWControl( ) const;
Returns a copy of the **UOW control**.

void setUOWControl(const MQLONG *control* **= MQCUOWC_ONLY);**
Sets the **UOW control**.

MQLONG version( ) const;
Returns the **version** number.

ImqBoolean setVersion(const MQLONG *version* **= MQCIH_VERSION_2);**
Sets the **version** number. It returns TRUE if successful.

Object data (protected)

MQLONG *olVersion*
The maximum MQCIH version number that can be accommodated in the storage allocated for *opcih*.

PMQCIH *opcih*
The address of an MQCIH data structure. The amount of storage allocated is indicated by *olVersion*.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR

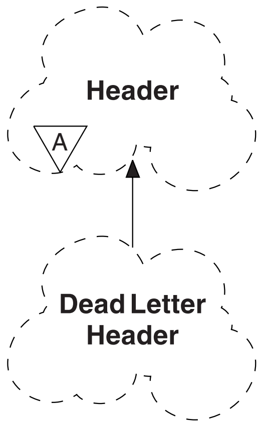
MQRC_WRONG_VERSION

Return codes

Table 3. ImqCICSBridgeHeader class return codes

Return Code	**Function**	**CompCode**	**Reason**	**Abend Code**
MQCRC_OK				
MQCRC_BRIDGE_ERROR			MQFB_CICS	
MQCRC_MQ_API_ERROR	MQSeries call name	MQSeries CompCode	MQSeries Reason	
MQCRC_BRIDGE_TIMEOUT	MQSeries call name	MQSeries CompCode	MQSeries Reason	
MQCRC_CICS_EXEC_ERROR	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_SECURITY_ERROR	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_PROGRAM_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_TRANSID_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	
MQCRC_BRIDGE_ABEND				CICS ABCODE
MQCRC_APPLICATION_ABEND				CICS ABCODE

ImqDeadLetterHeader



This class encapsulates specific features of the MQDLH data structure (see Table 9 on page 123). 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 50)
ImqItem (see “ImqItem” on page 55)
ImqMessage (see “ImqMessage” on page 57)
ImqString (see “ImqString” on page 105)

Object attributes

dead-letter reason code
The reason the message arrived on the dead-letter queue. The initial value is MQRC_NONE.

destination queue manager name
| The name of the original destination queue manager. The name is a string
| of length MQ_Q_MGR_NAME_LENGTH. Its initial value is null.

destination queue name
| The name of the original destination queue. The name is a string of length
| MQ_Q_NAME_LENGTH. Its initial value is null.

put application name
The name of the application that put the message on the dead-letter queue.
| The name is a string of length MQ_PUT_APPL_NAME_LENGTH. Its initial value is null.

put application type
The type of application that put the message on the dead-letter queue. The initial value is zero.

put date
| The date when the message was put on the dead-letter queue. The date is
| a string of length MQ_PUT_DATE_LENGTH. Its initial value is a null string.

put time
: The time when the message was put on the dead-letter queue. The time is a string of length MQ_PUT_TIME_LENGTH. Its initial value is a null string.

Constructors

ImqDeadLetterHeader( );
: The default constructor.

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

Overloaded "ImqItem" methods

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

  See the ImqHeader class method description on page 50 for further details.

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

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

  See the ImqHeader class method description on page 50 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**, stripped of any trailing blanks.

void setDestinationQueueManagerName(const char * *name* **);**
: Sets the **destination queue manager name**. Data longer than MQ_Q_MGR_NAME_LENGTH (48 characters) is truncated.

ImqString destinationQueueName( ) const ;
: Returns a copy of the **destination queue name**, stripped of any trailing blanks.

void setDestinationQueueName(const char * *name* **);**
: Sets the **destination queue name**. Data longer than MQ_Q_NAME_LENGTH (48 characters) is truncated.

ImqString putApplicationName( ) const ;
: Returns a copy of the **put application name**, stripped of any trailing blanks.

void setPutApplicationName(const char * *name* = 0);
: Sets the **put application name**. Data longer than MQ_PUT_APPL_NAME_LENGTH (28 characters) is truncated.

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**. Data longer than MQ_PUT_DATE_LENGTH (8 characters) is truncated.

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

void setPutTime(const char * *time* = 0);
: Sets the **put time**. Data longer than MQ_PUT_TIME_LENGTH (8 characters) is truncated.

Object data (protected)

MQDLH *omqdlh*
: The MQDLH data structure.

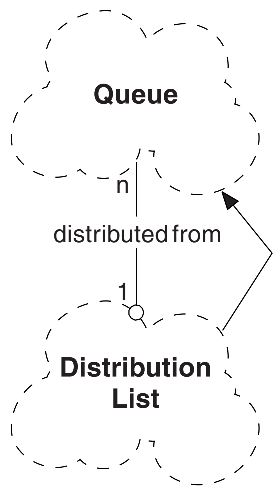
Reason codes

MQRC_INCONSISTENT_FORMAT

MQRC_STRUC_ID_ERROR

MQRC_ENCODING_ERROR

ImqDistributionList



This class encapsulates a dynamic distribution list that references one or more queues for the purpose of sending a message or messages to multiple destinations.

Other relevant classes

ImqMessage (see "ImqMessage" on page 57)

ImqQueue (see "ImqQueue" on page 82)

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 there are no such objects. To open an ImqDistributionList successfully, there must be at least one such object.

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

Constructors

ImqDistributionList( );
The default constructor.

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

Object methods (public)

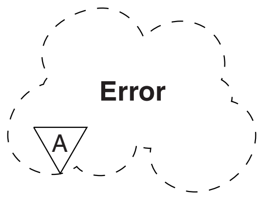
void operator = (const ImqDistributionList & *list* **);**
All ImqQueue objects that reference **this** object are dereferenced prior to copying. No ImqQueue objects will reference **this** object after the invocation of this method.

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. It relates to the MQI calls listed in Table 10 on page 123.

Other relevant classes

None.

Object attributes

completion code
The most recent completion code. The initial value is zero. The following additional values are possible:

MQCC_OK

MQCC_WARNING

MQCC_FAILED

reason code
The most recent reason code. The initial value is zero.

Constructors

ImqError( );
The default constructor.

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

Object methods (public)

void operator = (const ImqError & *error* **);**
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, and returns TRUE if successful.

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

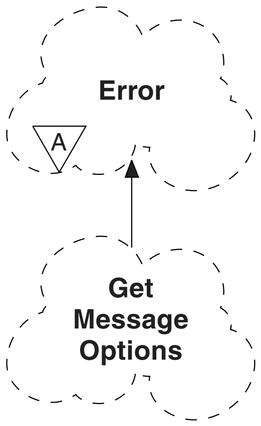
void setCompletionCode(const MQLONG *code* **= 0);**
: Sets the **completion code**.

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

Reason codes

MQRC_BUFFER_ERROR

ImqGetMessageOptions



This class encapsulates the MQGMO data structure (see Table 11 on page 123).

Other relevant classes

ImqString (see “ImqString” on page 105)

Object attributes

group status
Status of a message with respect to a group of messages. The initial value is MQGS_NOT_IN_GROUP. The following additional values are possible:

MQGS_MSG_IN_GROUP

MQGS_LAST_MSG_IN_GROUP

match options
Options for selecting incoming messages. The initial value is MQMO_MATCH_MSG_ID | MQMO_MATCH_CORREL_ID. The following additional values are possible:

MQMO_GROUP_ID

MQMO_MATCH_MSG_SEQ_NUMBER

MQMO_MATCH_OFFSET

MQMO_MSG_TOKEN

MQMO_NONE

message token
Message token. A binary value (MQBYTE16) of length MQ_MSG_TOKEN_LENGTH. The initial value is MQMTOK_NONE.

options
Options applicable to a message. The initial value is MQGMO_NO_WAIT. The following additional values are possible:

MQGMO_WAIT

MQGMO_SYNCPOINT

MQGMO_SYNCPOINT_IF_PERSISTENT

MQGMO_NO_SYNCPOINT

MQGMO_MARK_SKIP_BACKOUT

MQGMO_BROWSE_FIRST

MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_MSG_UNDER_CURSOR
MQGMO_LOCK
MQGMO_UNLOCK
MQGMO_ACCEPT_TRUNCATED_MSG
MQGMO_SET_SIGNAL
MQGMO_FAIL_IF_QUIESCING
MQGMO_CONVERT
MQGMO_LOGICAL_ORDER
MQGMO_COMPLETE_MSG
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE
MQGMO_NONE

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

returned length
Returned length. The initial value is MQRL_UNDEFINED. This attribute is read-only.

segmentation
The capability for segmentation of a message. The initial value is MQSEG_INHIBITED. The additional value, MQSEG_ALLOWED, is possible.

segment status
The segmentation status of a message. The initial value is MQSS_NOT_A_SEGMENT. The following additional values are possible:

MQSS_SEGMENT
MQSS_LAST_SEGMENT

syncpoint participation
TRUE when messages are retrieved under syncpoint control.

wait interval
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. The additional value, MQWI_UNLIMITED, is possible. This attribute is ignored unless the **options** include MQGMO_WAIT.

Constructors

ImqGetMessageOptions( );
The default constructor.

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

Object methods (public)

void operator = (const ImqGetMessageOptions & *gmo* **);**
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**.

ImqBinary messageToken( ) const;
Returns the **message token**.

ImqBoolean setMessageToken(const ImqBinary & *token* **);**
Sets the **message token**. The **data length** of *token* must be either zero or MQ_MSG_TOKEN_LENGTH. This method returns TRUE if successful.

void setMessageToken(const MQBYTE16 *token* **= 0);**
Sets the **message token**. *token* may be zero, which is the same as specifying MQMTOK_NONE. If *token* is nonzero, then it must address MQ_MSG_TOKEN_LENGTH bytes of binary data.

When using predefined values, such as MQMTOK_NONE, it may be necessary to make a cast to ensure a signature match, for example (MQBYTE *)MQMTOK_NONE.

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

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

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

MQLONG returnedLength( ) const;
Returns the **returned length**.

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

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

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

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

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

void setSyncPointParticipation(const ImqBoolean *sync* **);**
Sets the **syncpoint participation** value. If *sync* is TRUE, 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*
An MQGMO Version 2 data structure. Take care to access MQGMO fields supported for MQGMO_VERSION_2 only.

| This instance data is available for programs compiled on all MQSeries
| Version 5.1 products.

PMQGMO *opgmo*
The address of an MQGMO data structure. The version number for this address is indicated in *olVersion*. Take care to inspect the version number before accessing MQGMO fields, to ensure they are present.

This instance data is available for programs compiled on OS/390, and all MQSeries Version 5.1 products.

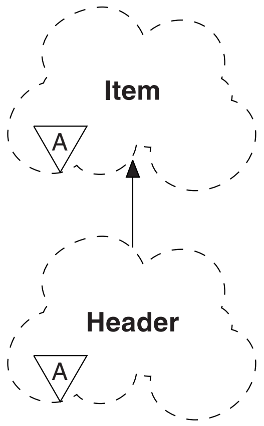
MQLONG *olVersion*
The version number of the MQGMO data structure addressed by *opgmo*.

This instance data is available for programs compiled on OS/390, and all MQSeries Version 5.1 products.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR

ImqHeader



This abstract class encapsulates common features of the MQDLH data structure (see Table 12 on page 124).

Other relevant classes

ImqCICSBridgeHeader (see "ImqCICSBridgeHeader" on page 32)
ImqDeadLetterHeader (see "ImqDeadLetterHeader" on page 39)
ImqIMSBridgeHeader (see "ImqIMSBridgeHeader" on page 52)
ImqItem (see "ImqItem" on page 55)
ImqMessage (see "ImqMessage" on page 57)
ImqReferenceHeader (see "ImqReferenceHeader" on page 102)
ImqString (see "ImqString" on page 105)
ImqWorkHeader (see "ImqWorkHeader" on page 114)

Object attributes

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

encoding
The original encoding. Initially MQENC_NATIVE.

format
The original format. Initially MQFMT_NONE.

header flags
The initial values are:

Zero for objects of the ImqDeadLetterHeader class
MQIIH_NONE for objects of the ImqIMSBridgeHeader class
MQRMHF_LAST for objects of the ImqReferenceHeader class
MQCIH_NONE for objects of the ImqCICSBridgeHeader class
MQWIH_NONE for objects of the ImqWorkHeader class

Constructors

ImqHeader( );
The default constructor.

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

Object methods (public)

void operator = (const ImqHeader & *header* **);**
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.

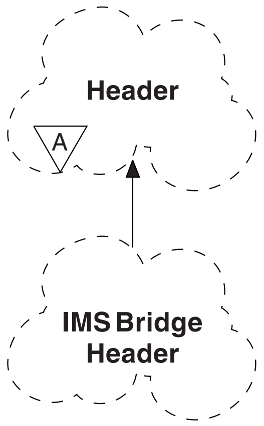
virtual void setFormat(const char * *name* **= 0);**
Sets the **format**, padding to 8 characters with trailing blanks.

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 (see Table 13



on page 124). Objects of this class are used by applications that send messages to the IMS bridge through MQSeries for OS/390.

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 27)

ImqHeader (see "ImqHeader" on page 50)

ImqItem (see "ImqItem" on page 55)

ImqMessage (see "ImqMessage" on page 57)

ImqString (see "ImqString" on page 105)

Object attributes

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

| **commit mode**
| Commit mode. See the *OTMA User's Guide* for more information about IMS
| commit modes. The initial value is MQICM_COMMIT_THEN_SEND. The
| additional value, MQICM_SEND_THEN_COMMIT, is possible.

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

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

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

| **security scope**
| Indicates the desired IMS security processing. The initial value is
| MQISS_CHECK. The additional value, MQISS_FULL, is possible.

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

| **transaction state**
| Indicates the state of the IMS conversation. The initial value is
| MQITS_NOT_IN_CONVERSATION. The additional value,
| MQITS_IN_CONVERSATION, is possible.

Constructors

ImqIMSBridgeHeader( );
The default constructor.

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

Overloaded "ImqItem" methods

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

See the parent class method description for further details.

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

To be successful, the **encoding** of the *msg* object 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. This method 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 nonzero, then it must address MQ_TRAN_INSTANCE_ID_LENGTH bytes of binary data. When using predefined 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**.

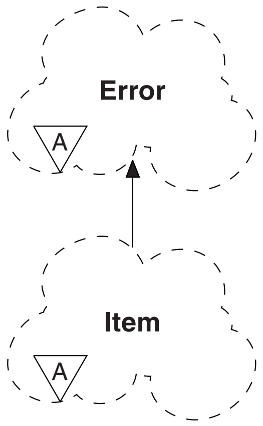
Object data (protected)

MQIIH *omqiih*
The MQIIH data structure.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR
MQRC_INCONSISTENT_FORMAT
MQRC_ENCODING_ERROR
MQRC_STRUC_ID_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.

| This class relates to the MQI calls listed in Table 14 on page 124.

Other relevant classes

ImqCache (see “ImqCache” on page 29)
ImqError (see “ImqError” on page 44)
ImqMessage (see “ImqMessage” on page 57)

Object attributes

structure id
A string of four characters at the beginning of the data structure. This
| attribute is read-only. This attribute is recommended for derived classes. It
| is not included automatically.

Constructors

ImqItem( );
The default constructor.

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

Class methods (public)

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

the message buffer currently addressed by the ImqCache **data pointer**. This method relies on the **structure id** and therefore is not guaranteed to work for all ImqItem derived classes.

Object methods (public)

void operator = (const ImqItem & *item* **);**
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. This method returns TRUE if successful.

Override this method to work with a specific subclass.

virtual ImqBoolean pasteIn(ImqMessage & *msg* **) = 0 ;**
Reads this object destructively[3] 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** including MQGMO_CONVERT.

If the read operation is successful, the ImqCache **data length** is reduced. This method returns TRUE if successful.

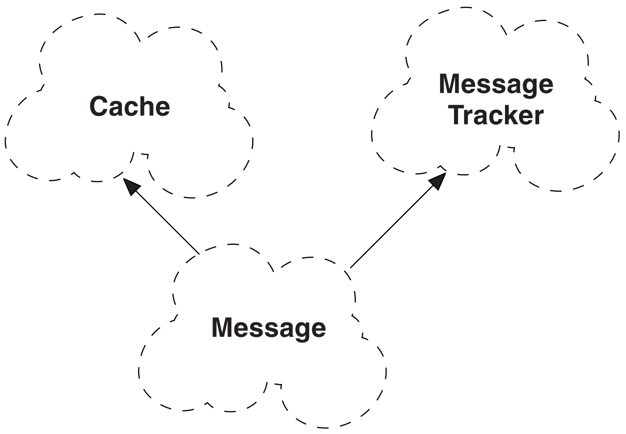
Override this method to work with a specific subclass.

Reason codes

MQRC_ENCODING_ERROR
MQRC_STRUC_ID_ERROR
MQRC_INCONSISTENT_FORMAT
MQRC_INSUFFICIENT_BUFFER
MQRC_INSUFFICIENT_DATA

3. 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 Table 15 on page 124), and also handles the construction and reconstruction of message data.

Other relevant classes

ImqCache (see "ImqCache" on page 29)

ImqItem (see "ImqItem" on page 55)

ImqMessageTracker (see "ImqMessageTracker" on page 64)

ImqString (see "ImqString" on page 105)

Object attributes

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

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

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

| **character set**
| Coded Character Set Id. The initial value is MQCCSI_Q_MGR. The
| following additional values are possible:
| MQCCSI_INHERIT
| MQCCSI_EMBEDDED

| You can also use a Coded Character Set Id of your choice. For information
| about this, see the code page conversion tables in the *MQSeries Application*
| *Programming Reference* book.

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 eight characters are truncated to eight characters. Names are always padded with blanks to eight characters. The initial value is MQFMT_NONE. The following additional values are possible:

MQFMT_ADMIN

MQFMT_CICS

MQFMT_COMMAND_1

MQFMT_COMMAND_2

MQFMT_DEAD_LETTER_HEADER

MQFMT_DIST_HEADER

MQFMT_EVENT

MQFMT_IMS

MQFMT_IMS_VAR_STRING

MQFMT_MD_EXTENSION

MQFMT_PCF

MQFMT_REF_MSG_HEADER

MQFMT_RF_HEADER

MQFMT_STRING

MQFMT_TRIGGER

MQFMT_WORK_INFO_HEADER

MQFMT_XMIT_Q_HEADER

You can also use an application-specific string of your choice. For more information about this, see the *Format* field of the message descriptor (MQMD) in the *MQSeries Application Programming Reference* book.

message flags
Segmentation control information. The initial value is MQMF_SEGMENTATION_INHIBITED. The following additional values are possible:

MQMF_SEGMENTATION_ALLOWED

MQMF_MSG_IN_GROUP

MQMF_LAST_MSG_IN_GROUP

MQMF_SEGMENT

MQMF_LAST_SEGMENT

MQMF_NONE

This attribute is not supported on OS/390.

message type
The broad categorization of a message. The initial value is MQMT_DATAGRAM. The following additional values are possible:

MQMT_SYSTEM_FIRST

MQMT_SYSTEM_LAST

MQMT_DATAGRAM

MQMT_REQUEST

MQMT_REPLY

MQMT_REPORT

MQMT_APPL_FIRST

MQMT_APPL_LAST

You can also use an application-specific string of your choice. For more information about this, see the *MsgType* field of the message descriptor (MQMD) in the *MQSeries Application Programming Reference* book.

offset Offset information. The initial value is zero. This attribute is not supported on OS/390.

original length
The original length of a segmented message. The initial value is MQOL_UNDEFINED. This attribute is not supported on OS/390.

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

MQPER_PERSISTENT

MQPER_NOT_PERSISTENT

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

put application name
The name of the application that put a message. The initial value is a null string.

put application type
The type of application that put a message. The initial value is MQAT_NO_CONTEXT. The following additional values are possible:

MQAT_AIX

MQAT_CICS

MQAT_CICS_BRIDGE

MQAT_DOS

MQAT_IMS

MQAT_IMS_BRIDGE

MQAT_MVS

MQAT_NOTES_AGENT

MQAT_OS2

MQAT_OS390

MQAT_OS400

MQAT_QMGR

MQAT_UNIX

MQAT_WINDOWS

MQAT_WINDOWS_NT

MQAT_XCF
MQAT_DEFAULT
MQAT_UNKNOWN
MQAT_USER_FIRST
MQAT_USER_LAST

You can also use an application-specific string of your choice. For more information about this, see the *PutApplType* field of the message descriptor (MQMD) in the *MQSeries Application Programming Reference* book.

put date
The date on which a message was put. The initial value is a null string.

put time
The time at which a message was put. The initial value is a null string.

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

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

report Feedback information associated with a message. The initial value is MQRO_NONE. The following additional values are possible:

MQRO_EXCEPTION
MQRO_EXCEPTION_WITH_DATA
MQRO_EXCEPTION_WITH_FULL_DATA *
MQRO_EXPIRATION
MQRO_EXPIRATION_WITH_DATA
MQRO_EXPIRATION_WITH_FULL_DATA *
MQRO_COA
MQRO_COA_WITH_DATA
MQRO_COA_WITH_FULL_DATA *
MQRO_COD
MQRO_COD_WITH_DATA
MQRO_COD_WITH_FULL_DATA *
MQRO_PAN
MQRO_NAN
MQRO_NEW_MSG_ID
MQRO_NEW_PASS_ID
MQRO_COPY_MSG_ID_TO_CORREL_ID
MQRO_PASS_CORREL_ID
MQRO_DEAD_LETTER_Q
MQRO_DISCARD_MSG
MQRO_DISCARD_MSG

where * indicates values that are not supported on MQSeries for OS/390.

sequence number
Sequence information identifying a message within a group. The initial value is one. This attribute is not supported on OS/390.

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

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

user id
: A user identity associated with a message. The initial value is a null string.

Constructors

ImqMessage( );
: The default constructor.

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

Object methods (public)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

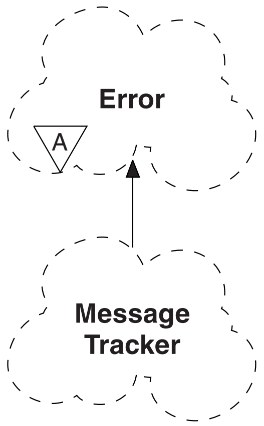
void setUserId(const char * *id* **= 0);**
Sets the **user id**.

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

Object data (protected)

MQMD *omqmd*
The MQMD data structure.

ImqMessageTracker



This class encapsulates those attributes of an ImqMessage or ImqQueue object that can be associated with either object. It relates to the MQI calls listed in Table 16 on page 125.

Other relevant classes

ImqBinary (see “ImqBinary” on page 27)

ImqError (see “ImqError” on page 44)

ImqMessage (see “ImqMessage” on page 57)

ImqQueue (see “ImqQueue” on page 82)

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. The additional value, MQCI_NEW_SESSION, is possible.

feedback
Feedback information to be sent with a message. The initial value is MQFB_NONE. The following additional values are possible:

MQFB_SYSTEM_FIRST

MQFB_SYSTEM_LAST

MQFB_APPL_FIRST

MQFB_APPL_LAST

MQFB_COA

MQFB_COD

MQFB_EXPIRATION

MQFB_PAN

MQFB_NAN

MQFB_QUIT
MQFB_DATA_LENGTH_ZERO
MQFB_DATA_LENGTH_NEGATIVE
MQFB_DATA_LENGTH_TOO_BIG
MQFB_BUFFER_OVERFLOW
MQFB_LENGTH_OFF_BY_ONE
MQFB_IIH_ERROR
MQFB_NOT_AUTHORIZED_FOR_IMS
MQFB_IMS_ERROR
MQFB_IMS_FIRST
MQFB_IMS_LAST
MQFB_CICS_APPL_ABENDED
MQFB_CICS_APPL_NOT_STARTED
MQFB_CICS_BRIDGE_FAILURE
MQFB_CICS_CCSID_ERROR
MQFB_CICS_CIH_ERROR
MQFB_CICS_COMMAREA_ERROR
MQFB_CICS_CORREL_ID_ERROR
MQFB_CICS_DLQ_ERROR
MQFB_CICS_ENCODING_ERROR
MQFB_CICS_INTERNAL_ERROR
MQFB_CICS_NOT_AUTHORIZED
MQFB_CICS_UOW_BACKED_OUT
MQFB_CICS_UOW_ERROR

You can also use an application-specific string of your choice. For more information about this, see the *Feedback* field of the message descriptor (MQMD) in the *MQSeries Application Programming Reference* book.

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. This method 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 nonzero, 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. This method 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 nonzero, then it must address MQ_CORREL_ID_LENGTH bytes of binary data. When using predefined 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. This method 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 nonzero, it must address MQ_GROUP_ID_LENGTH bytes of binary data. When using predefined 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**.

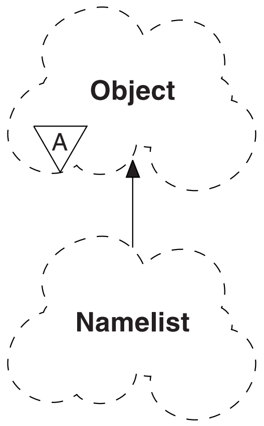
ImqBoolean setMessageId(const ImqBinary & *token* **);**
Sets the **message id**. The **data length** of *token* must be either zero or MQ_MSG_ID_LENGTH. This method returns TRUE if successful.

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 nonzero, it must address MQ_MSG_ID_LENGTH bytes of binary data. When using predefined 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

ImqNamelist



This class encapsulates a namelist. It relates to the MQI calls listed in Table 17 on page 125.

Other relevant classes

ImqObject (see "ImqObject" on page 70)

ImqString (see "ImqString" on page 105)

Object attributes

name count
The number of object names in **namelist names**. This attribute is read-only.

namelist names
Object names, the number of which is indicated by the **name count**. This attribute is read-only.

Constructors

ImqNamelist( );
The default constructor.

ImqNamelist(const ImqNamelist & *list* **);**
The copy constructor. The ImqObject **open status** will be false.

ImqNamelist(const char * *name***);**
Sets the ImqObject name to **name**.

Object methods (public)

void operator = (const ImqNamelist & *list* **);**
Instance data is copied from *list*, replacing the existing instance data. The ImqObject **open status** will be false.

ImqBoolean nameCount(MQLONG & *count* **);**
Provides a copy of the **name count**. It returns TRUE if successful.

MQLONG nameCount ( );
Returns the **name count** without any indication of possible errors.

ImqBoolean namelistName (const MQLONG *index***, ImqString &** *name* **);**
Provides a copy of one the **namelist names** by zero based index. It returns TRUE if successful.

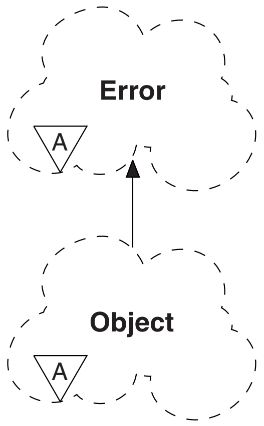
ImqString namelistName (const MQLONG *index* **);**
Returns one of the **namelist names** by zero based index without any indication of possible errors.

Reason codes

MQRC_INDEX_ERROR

MQRC_INDEX_NOT_PRESENT

ImqObject



This class is abstract. When an object of this class is destroyed, it is automatically closed, and its ImqQueueManager connection severed. This class relates to the MQI calls listed in Table 18 on page 125.

Other relevant classes

ImqBinary (see “ImqBinary” on page 27)

ImqError (see “ImqError” on page 44)

ImqNamelist (see “ImqNamelist” on page 68)

ImqQueue (see “ImqQueue” on page 82)

ImqQueueManager (see “ImqQueueManager” on page 94)

ImqString (see “ImqString” on page 105)

Class attributes

behavior
Controls the behavior of implicit opening.

IMQ_IMPL_OPEN (8L)
Implicit opening is allowed. This is the default.

Object attributes

alteration date
The alteration date. This attribute is read-only.

alteration time
The alteration time. This attribute is read-only.

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

alternate security id
The alternate security id. A binary value (MQBYTE40) of length MQ_SECURITY_ID_LENGTH. The initial value is MQSID_NONE.

close options
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. The initial value is zero.

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

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

name The name (up to 48 characters) of the queue manager, queue, namelist, or process, 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.

Note: An ImqQueueManager can have a null name, representing the default queue manager. The name changes to the actual queue manager after a successful **open**. An ImqDistributionList is dynamic and must have a null name.

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

open options
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 closes 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 121). 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 problems are likely to occur (see “Reopen” on page 19).

   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*).

Notes:

1. MQOO_OUTPUT is substituted for MQOO_INQUIRE during the **open** method for a distribution list, as MQOO_OUTPUT is the only valid **open option** at this time. However, this substitution may not occur in any future release, so you are recommended to set MQOO_OUTPUT explicitly in application programs that use the **open** method.

2. MQOO_RESOLVE_NAMES is an option that can be specified if the **resolved queue manager name** and **resolved queue name** attributes of the ImqQueue class are of interest.

open status
Indicates whether the object is open (TRUE) or closed (FALSE). The initial value is FALSE. This attribute is read-only.

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

queue manager identifier
Queue manager identifier. This attribute is read-only.

Constructors

ImqObject( );
The default constructor.

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

Class methods (public)

static MQLONG behavior( );
Returns the **behavior**.

void setBehavior(const MQLONG *behavior* **= 0);**
Sets the **behavior**.

Object methods (public)

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

ImqBoolean alterationDate(ImqString & *date* **);**
Provides a copy of the **alteration date**. It returns TRUE if successful.

ImqString alterationDate( );
Returns the **alteration date** without any indication of possible errors.

ImqBoolean alterationTime(ImqString & *time* **);**
Provides a copy of the **alteration time**. It returns TRUE if successful.

ImqString alterationTime( );
Returns the **alteration time** without any indication of possible errors.

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

ImqBoolean setAlternateUserId(const char * *id* **);**
Sets the **alternate user id**. The **alternate user id** can be set only while the **open status** is FALSE. This method returns TRUE if successful.

ImqBinary alternateSecurityId( ) const ;
Returns a copy of the **alternate security id.**

ImqBoolean setAlternateSecurityId(const ImqBinary & *token* **);**
Sets the **alternate security id**. The **alternate security id** can be set only while the **open status** is FALSE. The data length of *token* must be either zero or MQ_SECURITY_ID_LENGTH. It returns TRUE if successful.

ImqBoolean setAlternateSecurityId(const MQBYTE32 *token* **= 0);**
Sets the **alternate security id**. *token* may be zero, which is the same as specifying MQSID_NONE. If *token* is nonzero, it must address MQ_SECURITY_ID_LENGTH bytes of binary data. When using predefined values such as MQSID_NONE, it may be necessary to make a cast to ensure signature match; for example, (MQBYTE *)MQSID_NONE.

The **alternate security id** can be set only while the **open status** is TRUE. It returns TRUE if successful.

void setAlternateSecurityId(const unsigned char * *id* **= 0);**
Sets the **alternate security id**.

ImqBoolean close( );
Sets the **open status** to FALSE. It returns TRUE if successful.

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

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

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

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

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

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

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

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

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

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

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

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

ImqBoolean openFor(const MQLONG *required-options* **= 0);**
Attempts to ensure that the object is open with **open options** 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 are not changed; if the **open options** do not already contain any of the above, then MQOO_INPUT_AS_Q_DEF is set in the **open options**.

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

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

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

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

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

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

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

ImqBoolean queueManagerIdentifier(ImqString & *id* **);**
Provides a copy of the **queue manager identifier**. It returns TRUE if successful.

ImqString queueManagerIdentifier( );
Returns the **queue manager identifier** without any indication of possible errors.

Object methods (protected)

virtual ImqBoolean closeTemporarily( );
Closes an object safely prior to reopening. It 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 is 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, and 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 (valid only when **open status** is TRUE).

MQOD *omqod*
The embedded MQOD data structure. The amount of storage allocated for this data structure is that required for an MQOD Version 2. Inspect the version number *(omqod.Version)* and access the other fields as follows:

MQOD_VERSION_1
All other fields in *omqod* may be accessed.

MQOD_VERSION_2
All other fields in *omqod* may be accessed.

MQOD_VERSION_3
omqod.pmqod is a pointer to a dynamically allocated, larger, MQOD. No other fields in *omqod* may be accessed. All fields addressed by *omqod.pmqod* may be accessed.

Note: *omqod.pmqod.Version* may be less than *omqod.Version*, indicating that the MQSeries client has more functionality than the MQSeries server.

Reason codes

MQRC_ATTRIBUTE_LOCKED

MQRC_INCONSISTENT_OBJECT_STATE

MQRC_NO_CONNECTION_REFERENCE

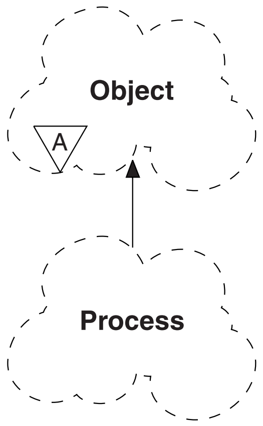
MQRC_STORAGE_NOT_AVAILABLE

MQRC_REOPEN_SAVED_CONTEXT_ERR

(reason codes from MQCLOSE)

(reason codes from MQCONN)
(reason codes from MQINQ)
(reason codes from MQOPEN)
(reason codes from MQSET)

ImqProcess



This class encapsulates an application process (an MQSeries object of type MQOT_PROCESS) that can be triggered by a trigger monitor (see Table 19 on page 126).

Other relevant classes

ImqObject (see “ImqObject” on page 70)

Object attributes

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

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

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

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

Constructors

ImqProcess( );
The default constructor.

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

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

Object methods (public)

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

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

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

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

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

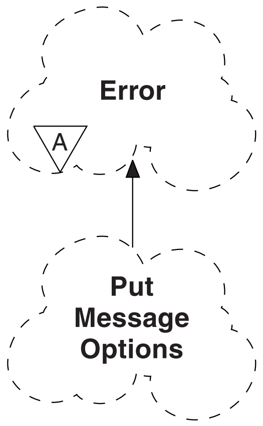
ImqBoolean environmentData(ImqString & *data* **);**
Provides a copy of the **environment data**. It returns TRUE if successful.

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

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

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

ImqPutMessageOptions



This class encapsulates the MQPMO data structure (see Table 20 on page 126).

Other relevant classes

ImqError (see “ImqError” on page 44)

ImqMessage (see “ImqMessage” on page 57)

ImqQueue (see “ImqQueue” on page 82)

ImqString (see “ImqString” on page 105)

Object attributes

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

options
The put message options. The initial value is MQPMO_NONE. The following additional values are possible:

- MQPMO_SYNCPOINT
- MQPMO_NO_SYNCPOINT
- MQPMO_NEW_MSG_ID
- MQPMO_NEW_CORREL_ID
- MQPMO_LOGICAL_ORDER
- MQPMO_NO_CONTEXT
- MQPMO_DEFAULT_CONTEXT
- MQPMO_PASS_IDENTITY_CONTEXT
- MQPMO_PASS_ALL_CONTEXT
- MQPMO_SET_IDENTITY_CONTEXT
- MQPMO_SET_ALL_CONTEXT
- MQPMO_ALTERNATE_USER_AUTHORITY
- MQPMO_FAIL_IF_QUIESCING

record fields
The flags that control the inclusion of put message records when a message is put. The initial value is MQPMRF_NONE. The following additional values are possible:

| MQPMRF_MSG_ID
| MQPMRF_CORREL_ID
| MQPMRF_GROUP_ID
| MQPMRF_FEEDBACK
| MQPMRF_ACCOUNTING_TOKEN

ImqMessageTracker attributes are taken from the 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
Name of a destination queue manager determined during a put. The initial value is null. This attribute is read-only.

resolved queue name
Name of a destination queue determined during a put. The initial value is null. This attribute is read-only.

syncpoint participation
TRUE when messages are put under syncpoint control.

Constructors

ImqPutMessageOptions( );
The default constructor.

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

Object methods (public)

void operator = (const ImqPutMessageOptions & *pmo* **);**
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 **syncpoint participation** value.

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

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

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

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

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

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

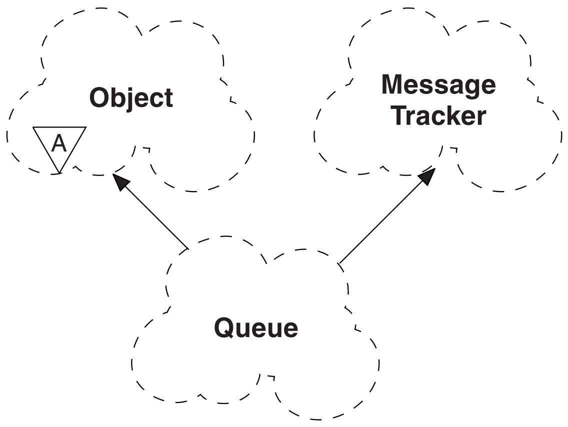
Object data (protected)

MQPMO *omqpmo*
The MQPMO data structure.

Reason codes

MQRC_STORAGE_NOT_AVAILABLE

ImqQueue



| This class encapsulates a message queue (an MQSeries object of type MQOT_Q). It
| relates to the MQI calls listed in Table 21 on page 126.

Other relevant classes

ImqCache (see "ImqCache" on page 29)
ImqDistributionList (see "ImqDistributionList" on page 42)
ImqGetMessageOptions (see "ImqGetMessageOptions" on page 46)
ImqMessage (see "ImqMessage" on page 57)
ImqMessageTracker (see "ImqMessageTracker" on page 64)
ImqObject (see "ImqObject" on page 70)
ImqPutMessageOptions (see "ImqPutMessageOptions" on page 79)
ImqQueueManager (see "ImqQueueManager" on page 94)
ImqString (see "ImqString" on page 105)

Object attributes

backout requeue name
Excessive backout requeue name. This attribute is read-only.

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

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

cluster name
Cluster name. This attribute is read-only.

cluster namelist name
Cluster namelist name. This attribute is read-only.

creation date
Queue creation data. This attribute is read-only.

creation time
Queue creation time. This attribute is read-only.

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

default bind
Default bind. This attribute is read-only.

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

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

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

definition type
Queue definition type. This attribute is read-only.

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

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

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

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

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

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

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

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

dynamic queue name
Dynamic queue name. The initial value is "AMQ.*" for all Personal Computer and UNIX platforms.

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

index type
Index type. This attribute is read-only.

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

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

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

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

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

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

next distributed queue
Next object of this class, in no particular order, having the same **distribution list reference** as this object. The initial value is zero.

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

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

previous distributed queue
Previous object of this class, in no particular order, having the same **distribution list reference** as this object. The initial value is zero.

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

queue manager name
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. The initial value is null.

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

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

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

resolved queue manager name
Resolved queue manager name. This attribute is read-only.

resolved queue name
Resolved queue name. This attribute is read-only.

retention interval
Queue retention interval. This attribute is read-only.

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

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

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

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

storage class
Storage class. This attribute is read-only.

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

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

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

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

trigger message priority
Threshold message priority for triggers. The initial value depends on the queue definition. This attribute is valid for a local queue only.

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

usage Usage. This attribute is read-only.

Constructors

ImqQueue( );
The default constructor.

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

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

Object methods (public)

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

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

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

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

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

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

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

ImqBoolean clusterName(ImqString & *name* **);**
Provides a copy of the **cluster name**. It returns TRUE if successful.

ImqString clusterName( );
Returns the **cluster name** without any indication of possible errors.

ImqBoolean clusterNamelistName(ImqString & *name* **);**
Provides a copy of the **cluster namelist name**. It returns TRUE if successful.

ImqString clusterNamelistName( );
Returns the **cluster namelist name** without any indication of errors.

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

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

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

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

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

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

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

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

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

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

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

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

ImqBoolean defaultBind(MQLONG & *bind* **);**
Provides a copy of the **default bind**. It returns TRUE if successful.

MQLONG defaultBind( );
Returns the **default bind** without any indication of possible errors.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ImqBoolean get(ImqMessage & *msg***, ImqGetMessageOptions &** *options* **);**
Retrieves a message from the queue, using the specified *options*. 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**, the buffer grows to accommodate any message retrieved. The **clearMessage** method is invoked against the *msg* object prior to retrieval.

This method returns TRUE if successful.

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

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

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

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

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

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

ImqBoolean indexType(MQLONG & *type* **);**
Provides a copy of the **index type**. It returns TRUE if successful.

MQLONG indexType( );
Returns the **index type** without any indication of possible errors.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This method returns TRUE if successful.

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

This method returns TRUE if successful.

Note: If the *pmo* includes a **context reference**, the referenced object 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 be set only while the ImqObject **open status** is FALSE. This method returns TRUE if successful.

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

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

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

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

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

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

ImqBoolean resolvedQueueManagerName(ImqString & *name* **);**
Provides a copy of the **resolved queue manager name**. It returns TRUE if successful.

Note: This method fails unless MQOO_RESOLVE_NAMES is among the ImqObject **open options**.

ImqString resolvedQueueManagerName( ) ;
Returns the **resolved queue manager name**, without any indication of possible errors.

ImqBoolean resolvedQueueName(ImqString & *name* **);**
Provides a copy of the **resolved queue name**. It returns TRUE if successful.

Note: This method fails unless MQOO_RESOLVE_NAMES is among the ImqObject **open options**.

ImqString resolvedQueueName( ) ;
Returns the **resolved queue name**, without any indication of possible errors.

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

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

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

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

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

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

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

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

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

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

ImqBoolean storageClass(ImqString & *class* **);**
Provides a copy of the **storage class**. It returns TRUE if successful.

ImqString storageClass( );
Returns the **storage class** without any indication of possible errors.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Object methods (protected)

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

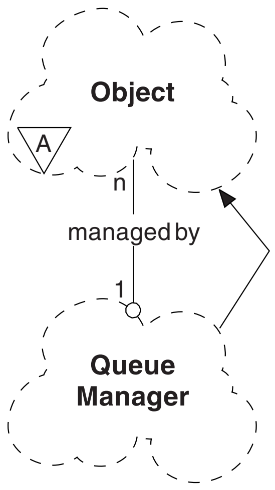
void setPreviousDistributedQueue(ImqQueue * *queue* **= 0);**
Sets the **previous distributed queue**.

Reason codes

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

(reason codes from MQPUT)

ImqQueueManager



This class encapsulates a queue manager (an MQSeries object of type MQOT_Q_MGR). It relates to the MQI calls listed in Table 22 on page 128.

Other relevant classes

ImqObject (see "ImqObject" on page 70)

Class attributes

behavior
Controls the behavior of implicit connection and disconnection.

IMQ_EXPL_DISC_BACKOUT (0L)
An explicit call to the **disconnect** method implies backout. This attribute is mutually exclusive with IMQ_EXPL_DISC_COMMIT.

IMQ_EXPL_DISC_COMMIT (1L)
An explicit call to the **disconnect** method implies commit (the default). This attribute is mutually exclusive with IMQ_EXPL_DISC_BACKOUT.

IMQ_IMPL_CONN (2L)
Implicit connection is allowed (the default).

IMQ_IMPL_DISC_BACKOUT (0L)
An implicit call to the **disconnect** method, which can occur during object destruction, implies backout. This attribute is mutually exclusive with the IMQ_IMPL_DISC_COMMIT.

IMQ_IMPL_DISC_COMMIT (4L)
An implicit call to the **disconnect** method, which can occur during object destruction, implies commit (the default). This attribute is mutually exclusive with IMQ_IMPL_DISC_BACKOUT.

Object attributes

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

begin options
: Options that apply to the **begin** method. The initial value is MQBO_NONE.

channel auto definition
: Channel auto definition value. This attribute is read-only.

channel auto definition event
: Channel auto definition event value. This attribute is read-only.

channel auto definition exit
: Channel auto definition exit name. This attribute is read-only.

character set
: Coded character set identifier (CCSID). This attribute is read-only.

cluster workload data
: Cluster workload exit data. This attribute is read-only.

cluster workload exit
: Cluster workload exit name. This attribute is read-only.

cluster workload length
: Cluster workload length. This attribute is read-only.

command input queue name
: System command input queue name. This attribute is read-only.

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

connect options
: Options that apply to the **connect** method. The initial value is MQCNO_NONE. The following additional values are possible:

  MQCNO_STANDARD_BINDING

  MQCNO_FASTPATH_BINDING

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

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

default transmission queue name
: Default transmission queue name. This attribute is read-only.

distribution lists
: Specifies 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. The initial value is zero.

inhibit event
: Controls inhibit events. This attribute is read-only.

local event
: Controls local events. This attribute is read-only.

maximum handles
: Maximum number of handles. This attribute is read-only.

maximum message length
Maximum possible length for any message on any queue managed by this queue manager. This attribute is read-only.

maximum priority
Maximum message priority. This attribute is read-only.

maximum uncommitted messages
This is the maximum number of uncommitted messages within a unit or work. This attribute is read-only.

performance event
Controls performance events. This attribute is read-only.

platform
Platform on which the queue manager resides. This attribute is read-only.

remote event
Controls remote events. This attribute is read-only.

| **repository name**
| Repository name. This attribute is read-only.

| **repository namelist**
| Name of the repository namelist. This attribute is read-only.

start-stop event
Controls start-stop events. This attribute is read-only.

syncpoint availability
Availability[4] of syncpoint participation. This attribute is read-only.

trigger interval
Trigger interval. This attribute is read-only.

Constructors

ImqQueueManager( );
The default constructor.

ImqQueueManager(const ImqQueueManager & *manager* **);**
The copy constructor. The **connection status** will be FALSE.

ImqQueueManager(const char * *name* **);**
Sets the ImqObject **name** to *name*.

Destructors

When an ImqQueueManager object is destroyed, it is automatically disconnected.

Class methods (public)

static MQLONG behavior( );
Returns the **behavior**.

void setBehavior(const MQLONG *behavior* **= 0);**
Sets the **behavior**.

4. Queue manager-coordinated global units of work are not supported on the OS/400 platform. You can program a unit of work, externally coordinated by OS/400, using the "_Rcommit" and "_Rback" native system calls. This type of unit of work is started by starting the MQSeries application under job-level commitment control using the STRCMTCTL command. See the *MQSeries Application Programming Guide* for further details. **Backout** and **commit** are supported on the OS/400 platform for local units of work coordinated by a queue manager.

Object methods (public)

void operator = (const ImqQueueManager & *mgr* **);**
Disconnects if necessary, and copies instance data from *mgr*. The **connection status** is be FALSE.

ImqBoolean authorityEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **authority event**. It returns TRUE if successful.

MQLONG authorityEvent( );
Returns the enablement state of the **authority event** without any indication of possible errors.

ImqBoolean backout( );
Backs out uncommitted changes. It returns TRUE if successful.

ImqBoolean begin( );
Begins a unit of work. The **begin options** affect the behavior of this method. It returns TRUE if successful.

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

void setBeginOptions(const MQLONG *options* **= MQBO_NONE);**
Sets the **begin options**.

ImqBoolean channelAutoDefinition(MQLONG & *value* **);**
Provides a copy of the **channel auto definition** value. It returns TRUE if successful.

MQLONG channelAutoDefinition( );
Returns the **channel auto definition** value without any indication of possible errors.

ImqBoolean channelAutoDefinitionEvent(MQLONG & *value* **);**
Provides a copy of the **channel auto definition event** value. It returns TRUE if successful.

MQLONG channelAutoDefinitionEvent( );
Returns the **channel auto definition event** value without any indication of possible errors.

ImqBoolean channelAutoDefinitionExit(ImqString & *name* **);**
Provides a copy of the **channel auto definition exit** name. It returns TRUE if successful.

ImqString channelAutoDefinitionExit( );
Returns the **channel auto definition exit** name without any indication of possible errors.

ImqBoolean characterSet(MQLONG & *ccsid* **);**
Provides a copy of the **character set**. It returns TRUE if successful.

MQLONG characterSet( );
Returns a copy of the **character set**, without any indication of possible errors.

| **ImqBoolean clusterWorkloadData(ImqString &** *data* **);**
| Provides a copy of the **cluster workload exit data**. It returns TRUE if
| successful.

ImqString clusterWorkloadData( );
Returns the **cluster workload exit data** without any indication of possible errors.

ImqBoolean clusterWorkloadExit(ImqString & *name* **);**
Provides a copy of the **cluster workload exit name**. It returns TRUE if successful.

ImqString clusterWorkloadExit( );
Returns the **cluster workload exit name** without any indication of possible errors.

ImqBoolean clusterWorkloadLength(MQLONG & *length* **);**
Provides a copy of the **cluster workload length**. It returns TRUE if successful.

MQLONG clusterWorkloadLength( );
Returns the **cluster workload length** without any indication of possible errors.

ImqBoolean commandInputQueueName(ImqString & *name* **);**
Provides a copy of the **command input queue name**. It returns TRUE if successful.

ImqString commandInputQueueName( );
Returns the **command input queue name** without any indication of possible errors.

ImqBoolean commandLevel(MQLONG & *level* **);**
Provides a copy of the **command level**. It returns TRUE if successful.

MQLONG commandLevel( );
Returns the **command level** without any indication of possible errors.

ImqBoolean commit( );
Commits uncommitted changes. It returns TRUE if successful.

ImqBoolean connect( );
Connects to the queue manager with the given ImqObject **name**, the default being the local queue manager. If you want to connect to a specific queue manager, use the ImqObject **setName** method before connection. The **connect options** affect the behavior of this method. This method sets the **connection status** to TRUE, and returns TRUE if successful.

Note: More than one ImqQueueManager object can be connected to the same queue manager. All use the same MQHCONN connection handle and share UOW functionality for the connection associated with the thread. The first ImqQueueManager to connect obtains the MQHCONN handle. The last ImqQueueManager to disconnect performs the MQDISC.

For a multithreaded program, each thread must use a separate ImqQueueManager object. Connections in different threads have different MQHCONN connection handles.

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**. It returns TRUE if successful.

ImqString deadLetterQueueName( );
Returns a copy of the **dead-letter queue name**, without any indication of possible errors.

ImqBoolean defaultTransmissionQueueName(ImqString & *name* **);**
Provides a copy of the **default transmission queue name**. It returns TRUE if successful.

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

ImqBoolean disconnect( );
Disconnects from the queue manager and sets the **connection status** to FALSE. 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, only the last to disconnect performs a physical disconnection; others perform a logical disconnection. Uncommitted changes are committed (on physical disconnection only). It returns TRUE if successful.

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

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

ImqObject * firstManagedObject( ) const ;
Returns the **first managed object**.

ImqBoolean inhibitEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **inhibit event**. It returns TRUE if successful.

MQLONG inhibitEvent( );
Returns the enablement state of the **inhibit event** without any indication of possible errors.

ImqBoolean localEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **local event**. It returns TRUE if successful.

MQLONG localEvent( );
Returns the enablement state of the **local event** without any indication of possible errors.

ImqBoolean maximumHandles(MQLONG & *number* **);**
Provides a copy of the **maximum handles**. It returns TRUE if successful.

MQLONG maximumHandles( );
Returns the **maximum handles** without any indication of possible errors.

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

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

ImqBoolean maximumPriority(MQLONG & *priority* **);**
Provides a copy of the **maximum priority**. It returns TRUE if successful.

MQLONG maximumPriority( );
Returns a copy of the **maximum priority**, without any indication of possible errors.

ImqBoolean maximumUncommittedMessages(MQLONG & *number* **);**
Provides a copy of the **maximum uncommitted messages**. It returns TRUE if successful.

MQLONG maximumUncommittedMessages( );
Returns the **maximum uncommitted messages** without any indication of possible errors.

ImqBoolean performanceEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **performance event**. It returns TRUE if successful.

MQLONG performanceEvent( );
Returns the enablement state of the **performance event** without any indication of possible errors.

ImqBoolean platform(MQLONG & *platform* **);**
Provides a copy of the **platform**. It returns TRUE if successful.

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

ImqBoolean remoteEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **remote event**. It returns TRUE if successful.

MQLONG remoteEvent( );
Returns the enablement state of the **remote event** without any indication of possible errors.

| **ImqBoolean repositoryName(ImqString &** *name* **);**
| Provides a copy of the **repository name**. It returns TRUE if successful.

| **ImqString repositoryName( );**
| Returns the **repository name** without any indication of possible errors.

| **ImqBoolean repositoryNamelistName(ImqString &** *name* **);**
| Provides a copy of the **repository namelist name**. It returns TRUE if
| successful.

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

ImqBoolean startStopEvent(MQLONG & *event* **);**
Provides a copy of the enablement state of the **start-stop event**. It returns TRUE if successful.

MQLONG startStopEvent( );
Returns the enablement state of the **start-stop event** without any indication of possible errors.

ImqBoolean syncPointAvailability(MQLONG & *sync* **);**
Provides a copy of the **syncpoint availability** value. It returns TRUE if successful.

MQLONG syncPointAvailability( );
Returns a copy of the **syncpoint availability** value, without any indication of possible errors.

ImqBoolean triggerInterval(MQLONG & *interval* **);**
Provides a copy of the **trigger interval**. It returns TRUE if successful.

MQLONG triggerInterval( );
Returns the **trigger interval** without any indication of possible errors.

Object methods (protected)

void setFirstManagedObject(const ImqObject * *object* **= 0);**
Sets the **first managed object**.

Object data (protected)

MQHCONN *ohconn*
The MQSeries connection handle (meaningful only while the **connection status** is TRUE).

Reason codes

MQRC_ENVIRONMENT_ERROR

(reason codes for MQBACK)

(reason codes for MQBEGIN)

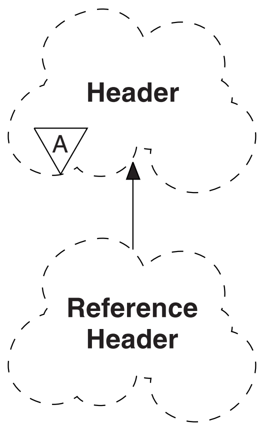
(reason codes for MQCMIT)

(reason codes for MQCONNX)

(reason codes for MQDISC)

(reason codes for MQCONN)

ImqReferenceHeader



| This class encapsulates specific features of the MQRMH data structure. It relates to
| the MQI calls listed in Table 23 on page 129.

Other relevant classes

ImqBinary (see "ImqBinary" on page 27)

ImqHeader (see "ImqHeader" on page 50)

ImqItem (see "ImqItem" on page 55)

ImqMessage (see "ImqMessage" on page 57)

ImqString (see "ImqString" on page 105)

Object attributes

destination environment
Environment for the destination. The initial value is a null string.

destination name
Name of the data destination. The initial value is a null string.

instance id
Binary value (MQBYTE24) of length MQ_OBJECT_INSTANCE_ID_LENGTH. The initial value is MQOII_NONE.

logical length
Logical, or intended, length of message data that follows this header. The initial value is zero.

logical offset
Logical offset for the message data that follows, to be interpreted in the context of the data as a whole, at the ultimate destination. The initial value is zero.

logical offset 2
High-order extension to the **logical offset**. The initial value is zero.

reference type
Reference type. The initial value is a null string.

source environment
Environment for the source. The initial value is a null string.

source name
Name of the data source. The initial value is a null string.

Constructors

ImqReferenceHeader( );
The default constructor.

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

Overloaded "ImqItem" methods

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

See the ImqHeader class method description on 50 for further details.

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

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

See the ImqHeader class method description on 50 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. This method 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 nonzero, 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**.

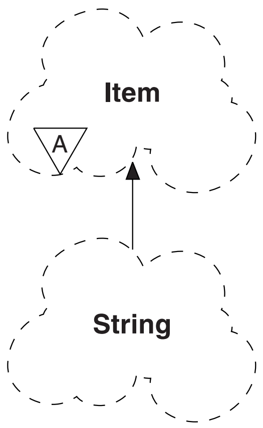
Object data (protected)

MQRMH *omqrmh*
The MQRMH data structure.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR
MQRC_STRUC_LENGTH_ERROR
MQRC_STRUC_ID_ERROR
MQRC_INSUFFICIENT_DATA
MQRC_INCONSISTENT_FORMAT
MQRC_ENCODING_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 55)

ImqMessage (see "ImqMessage" on page 57)

Object attributes

characters
: Characters in the **storage** which precede a trailing null.

length
: Number of bytes in the **characters**. If there is no **storage**, the **length** is zero. The initial value is zero.

storage
: A volatile array of bytes of arbitrary size. A trailing null must always be present in the **storage** after the **characters**, so that the end of the **characters** can be detected. Methods ensure that this situation is maintained, but 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** attribute.

Constructors

ImqString( );
: The default constructor.

ImqString(const ImqString & *string* **);**
: The copy constructor.

ImqString(const char *c* **);**
: The **characters** comprise *c*.

ImqString(const char * *text* **);**
: The **characters** are copied from *text*.

ImqString(const void * *buffer*, **const size_t** *length* **);**
: Copies *length* bytes starting from *buffer* and assigns them to the **characters**. Substitution is made for any null characters copied. The substitution

character is a period (.). No special consideration is given to any other non-printable or non-displayable characters copied.

Class methods (public)

static ImqBoolean copy(char * *destination-buffer***, const size_t** *length***, const char *** *source-buffer***, const char** *pad* **= 0);**
Copies up to *length* bytes from *source-buffer* to *destination-buffer*. If the number of characters in *source-buffer* is insufficient, 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. This method returns TRUE if successful.

Overloaded "ImqItem" methods

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

See the parent class method description for further details.

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

To be successful, the **encoding** of the *msg* object 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 substring by copying bytes from the **characters** starting at *offset*. If *length* is zero, the rest of the **characters** are returned. If the combination of *offset* and *length* does not produce a reference within the **characters**, 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" ;
"string one" + strTwo ;
```

Note: Although most compilers accept **strOne + "string two";** Microsoft® Visual C++ requires **strOne + (char *)"string two" ;**

ImqString operator + (const ImqString & *string1* **) const ;**
Returns the result of appending *string1* to the **characters**.

ImqString operator + (const double *number* **) const ;**
Returns the result of appending *number* to the **characters** after conversion to text.

ImqString operator + (const long *number* **) const ;**
Returns the result of appending *number* to the **characters** after conversion to text.

void operator += (const char *c* **);**
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**. This method 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. It returns either TRUE or FALSE.

short compare(const ImqString & *string* **) const ;**
Compares the **characters** with those of *string*. The result is zero if the **characters** are equal, negative if "less than" and positive if "greater than". Comparison is case sensitive. A null ImqString is regarded as "less than" a nonnull ImqString.

ImqBoolean copyOut(char * *buffer***, const size_t** *length***, const char** *pad* **= 0);**
Copies up to *length* bytes from the **characters** to the *buffer*. If the number of **characters** is insufficient, then the remaining space in *buffer* is filled with *pad* characters. *buffer* may be zero if *length* is also zero. It returns TRUE if successful.

size_t copyOut(long & *number* **) const ;**
Sets *number* from the **characters** after conversion from text, and returns the number of characters involved in the conversion. If this is zero, no conversion has been performed and *number* is not set. A convertible character sequence must begin with the values shown in Figure 15 on page 108.

```
<blank(s)>
<+|->
digit(s)
```

Figure 15. 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 from *c*, 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 string shown in Figure 16 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 16. 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 precede the *token* characters. If *c* is not a blank, characters *c* that directly succeed the *token* characters are also removed. The number of characters removed is returned. For example, the string shown in Figure 17 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 17. Retrieving tokens from string text

Figure 18 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 "C".
// strFile becomes "OS2", then "BITMAP",
// then "OS2LOGO.BMP" leaving strPath empty.
```

Figure 18. 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, it returns FALSE. Otherwise, it returns TRUE. If *string* is null, it returns TRUE.

ImqBoolean find(const ImqString & *string*, **size_t &** *offset* **);**
: Searches for an exact match for *string* somewhere within the **characters** from offset *offset* onwards. If *string* is null, it returns TRUE without updating *offset*. If no match is found, it returns FALSE; note that the value of *offset* may have been increased. If a match is found, it returns TRUE and updates *offset* to the offset of *string* within the **characters**.

size_t length( ) const ;
: Returns the **length**.

ImqBoolean pasteIn(const double *number*, **const char *** *format* **= "%f");**
: *number* is appended to the **characters** after conversion to text. It returns TRUE if successful.

The specification *format* is used to format the floating point conversion. If specified, it 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. It returns TRUE if successful.

ImqBoolean pasteIn(const void * *buffer*, **const size_t** *length* **);**
: Appends *length* bytes from *buffer* to the **characters**, and adds a final trailing null. A substitution is made for any null characters copied. The substitution character is a period (.). No special consideration is given to any other nonprintable or nondisplayable characters copied. This method returns TRUE if successful.

ImqBoolean set(const char * *buffer*, **const size_t** *length* **);**
: Sets the **characters** from a fixed-length character field, which might contain a null. A null is appended to the characters from the fixed-length field if necessary. This method 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.

This method 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. It returns TRUE if successful.

Reason codes

MQRC_DATA_TRUNCATED

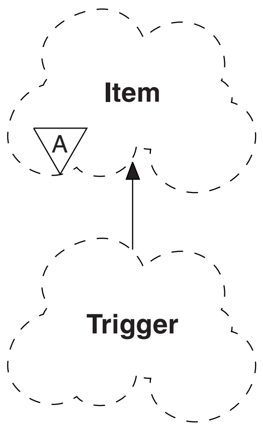
MQRC_NULL_POINTER

MQRC_STORAGE_NOT_AVAILABLE

MQRC_BUFFER_ERROR

MQRC_INCONSISTENT_FORMAT

ImqTrigger



This class encapsulates the MQTM data structure (see Table 24 on page 130). 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 46)
ImqItem (see "ImqItem" on page 55)
ImqMessage (see "ImqMessage" on page 57)
ImqString (see "ImqString" on page 105)

Object attributes

application id
Identity of the application that sent the message. The initial value is a null string.

application type
Type of application that sent the message. The initial value is zero. The following additional values are possible:

MQAT_AIX
MQAT_CICS
MQAT_DOS
MQAT_IMS
MQAT_MVS
MQAT_NOTES_AGENT
MQAT_OS2
MQAT_OS390
MQAT_OS400
MQAT_UNIX
MQAT_WINDOWS
MQAT_WINDOWS_NT

| MQAT_USER_FIRST
| MQAT_USER_LAST

| **environment data**
Environment data for the process. The initial value is a null string.

process name
Process name. The initial value is a null string.

queue name
Name of the queue to be started. The initial value is a null string.

trigger data
Trigger data for the process. The initial value is a null string.

user data
User data for the process. The initial value is a null string.

Constructors

ImqTrigger( );
The default constructor.

ImqTrigger(const ImqTrigger & *trigger* **);**
The copy constructor.

Overloaded "ImqItem" methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Writes an MQTM data structure to the message buffer, replacing any existing content. Sets the *msg* **format** to MQFMT_TRIGGER.

See the ImqItem class method description on 55 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 on 55 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. This method returns TRUE if successful.

ImqString environmentData( ) const ;
Returns a copy of the **environment data**.

void setEnvironmentData(const char * *data* **);**
Sets the **environment data**.

ImqString processName( ) const ;
Returns a copy of the **process name**.

void setProcessName(const char * *name* **);**
Sets the **process name**, padding with blanks to 48 characters.

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

void setQueueName(const char * *name* **);**
Sets the **queue name**, padding with blanks to 48 characters.

ImqString triggerData( ) const ;
Returns a copy of the **trigger data**.

void setTriggerData(const char * *data* **);**
Sets the **trigger data**.

ImqString userData( ) const ;
Returns a copy of the **user data**.

void setUserData(const char * *data* **);**
Sets the **user data**.

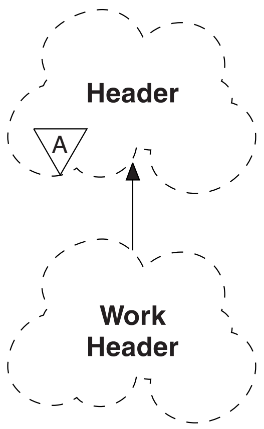
Object data (protected)

MQTM *omqtm*
The MQTM data structure.

Reason codes

MQRC_NULL_POINTER
MQRC_INCONSISTENT_FORMAT
MQRC_ENCODING_ERROR
MQRC_STRUC_ID_ERROR

ImqWorkHeader



This class encapsulates specific features of the MQWIH data structure (see Table 25 on page 130). Objects of this class are used by applications putting messages to the queue managed by the OS/390 Workload Manager.

Other relevant classes

ImqBinary (see "ImqBinary" on page 27)

ImqHeader (see "ImqHeader" on page 50)

ImqItem (see "ImqItem" on page 55)

ImqMessage (see "ImqMessage" on page 57)

ImqString (see "ImqString" on page 105)

Object attributes

message token
Message token for the OS/390 Workload Manager, of length MQ_MSG_TOKEN_LENGTH. The initial value is MQMTOK_NONE.

service name
The 32-character name of a process. The name is initially blanks.

service step
The 8-character name of a step within the process. The name is initially blanks.

Constructors

ImqWorkHeader( );
The default constructor.

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

Overloaded "ImqItem" methods

virtual ImqBoolean copyOut(ImqMessage & *msg* **);**
Inserts an MQWIH data structure into the beginning of the message buffer, moving the existing message data further along, and sets the *msg* **format** to MQFMT_WORK_INFO_HEADER.

See the parent class method description for more details.

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

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

The ImqMessage format must be MQFMT_WORK_INFO_HEADER.

See the parent class method description for more details.

Object methods (public)

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

ImqBinary messageToken ( ) const;
Returns the **message token**.

ImqBoolean setMessageToken(const ImqBinary & *token* **);**
Sets the **message token**. The data length of *token* must be either zero or MQ_MSG_TOKEN_LENGTH. It returns TRUE if successful.

void setMessageToken(const MQBYTE16 *token* **= 0);**
Sets the **message token**. *token* may be zero, which is the same as specifying MQMTOK_NONE. If *token* is nonzero, then it must address MQ_MSG_TOKEN_LENGTH bytes of binary data.

When using predefined values such as MQMTOK_NONE, it may be necessary to make a cast to ensure a signature match; for example, (MQBYTE *)MQMTOK_NONE.

ImqString serviceName ( ) const;
Returns the **service name**, including trailing blanks.

void setServiceName(const char * *name* **);**
Sets the **service name**.

ImqString serviceStep ( ) const;
Returns the **service step**, including trailing blanks.

void setServiceStep(const char * *step* **);**
Sets the **service step**.

Object data (protected)

MQWIH omqwih
The MQWIH data structure.

Reason codes

MQRC_BINARY_DATA_LENGTH_ERROR

Changes

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 AS/400” on page 118.

If you are writing programs for the Windows 95 and Windows NT platforms, see “Compiling VisualAge C++ sample programs for Windows 95, 98, and NT” on page 119.

If you are writing programs for the OS/390 platform, see “Building an application on OS/390” on page 119.

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 4. 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/mqm/inc	-limqb23ia[_r] -limq{c|s}23ia[_r] (see note 2)
AS/400	IBM ILE for C++ for AS/400	DFTCHAR*(SIGNED)	BNDSRVPGM(QMQM/IMQ B23I4[_R] QMQM/IMQS23I4[_R]) (see note 2)
HP-UX V10.20 and V11	HP C++ Version 3.1 on HP-UX V10.20 and HP C++ Version 12.0 on HP-UX V11	CC +eh -D_HPUX_SOURCE (see notes 3 and 4)	-limqb23hh[_r|_d] -limq{c|s}23hh[_r|_d] (see note 5)
	HP ANSII C++	aCC -D_HPUX_SOURCE (see note 4)	-limqi23ah[_r|_d] {-lmqm[_r|_d] | -lmqic[_r|_d]} (see note 6)
OS/2 Warp	IBM VisualAge for C++ Version 3.0 for OS/2	icc /Gd /Gm /Gs /J-	imqb23i2 imq{c|s}23i2
OS/390	IBM OS/390 C/C++ Version 2 Release 4 or later	/cxx	(see note 7)
Sun Solaris	Sun WorkShop Compiler C++ Version 4.2	CC -mt	-limqb23ss[_d] -limq{c|s}23ss[_d] {-lmqic|-lmqm} -lmqmcs[_d] -lmqmzse -lsocket -lnsl -ldl (see notes 6 and 8)
Windows 3.1 (16-bit client only)	Microsoft Visual C++ Version 1.5 for Windows 3.1	cl -ALw -Mq	imqb23vw imqc23vw mqic
Windows NT, Windows 95, and Windows 98	IBM VisualAge for C++ for Windows Version 3.5	icc /Gd /Gm /Gs /J-	imqb23in imq{c|s}23in
	Microsoft Visual C++ Version 4.0	cl -MD	imqb23vn imq{c|s}23vn

Table 4. MQSeries C++ switches and link libraries (continued)

Platform	Compiler	Switches	Libraries

Notes:

1. {c|s} denotes that you must enter either ‘c’ for a client application or ‘s’ for a server application.
2. To build a threaded application you need to link with the libraries ending in _r.
3. The link and run-time libraries supplied for HP-UX now cater for exception handling. Programs using the older imq{b|c|s}23ch[_r] libraries should be recompiled with the +eh option and relinked with the newer imq{b|c|s|}23hh[_r] libraries.
4. If you compile with CC, you must also link with CC. Similarly, if you compile with aCC, you must also link with aCC.
5. To build non-threaded applications, you do not need to link with libraries ending in _r or _d; the non-underscore version of the libraries suffices. To build draft 10 pthreads applications on HP-UX V11 and draft 4 pthreads applications on HP-UX V10.20, link with the libraries ending in _r. To build draft 4 pthreads applications on HP-UX V11, link with the libraries ending in _d.
6. Library -lmqm must be linked in with a server application whereas the library -lmqic must be linked in with a client application.
7. For OS/390, the side decks are imqs23dm imqb23dm, or imqs23dr imqb23dr, or imqs23dc imqb23dc. The corresponding DLL load modules are imqs23im imqb23im, or imqs23ir imqb23ir, or imqs23ic imqb23ic.
8. To build non-threaded applications, you do not need to link with libraries ending in _r or _d; the non-underscore version of the libraries suffices. To build a DCE (POSIX draft 4) threaded application, link with the libraries ending in _d.

Compiling C++ sample programs for AS/400

This section is aimed at the C++ programmer who wants to write programs for the OS/400® platform.

IBM ILE C++ for AS/400 (program 5799-GDW) is a native compiler for C++ programs. In addition, IBM VisualAge® for C++ for AS/400 provides cross-compilers with clients running on OS/2, Windows 95, or Windows NT. The cross-compilers also generate object modules that can be bound into OS/400 programs.

The following instructions describe how to use the native compiler to create MQSeries C++ applications. Users of the VisualAge cross-compilers should interpret the instructions according to their version of the product.

1. Install the ILE C++ for AS/400 compiler as directed in the *Read Me first!* manual that accompanies the product.
2. Ensure that the QCXXN library is in your library list.
3. Create the HELLO WORLD sample program:
   a. Create a module:
   ```
   CRTCPPMOD MODULE(MYLIB/IMQWRLD) +
   SRCSTMF('/QIBM/ProdData/mqm/samp/imqwrld.cpp') +
   INCDIR('/QIBM/ProdData/mqm/inc') DFTCHAR(*SIGNED)
   ```
   The source for the C++ sample programs can be found in /QIBM/ProdData/mqm/samp and the include files in /QIBM/ProdData/mqm/inc.
   b. Bind this with MQSeries-supplied service programs to produce a program object:
   ```
   CRTPGM PGM(MYLIB/IMQWRLD) MODULE(MYLIB/IMQWRLD) +
   BNDSRVPGM(QMQM/IMQB23I4 QMQM/IMQS23I4)
   ```

See Table 4 on page 117 for alternative service programs that can be used.

c. Execute the HELLO WORLD sample program, using SYSTEM.DEFAULT.LOCAL.QUEUE:

```
CALL PGM(MYLIB/IMQWRLD)
```

Compiling VisualAge C++ sample programs for Windows 95, 98, and NT

This section is aimed at the C++ programmer, who wishes to write VisualAge programs for the Windows 95, 98, and NT platforms.

The IBM VisualAge for C++ for Windows Version 3.5 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, type `applicname`

Building an application on OS/390

You can write C++ programs for three of the environments that MQSeries for OS/390 supports:

- Batch
- RRS batch
- CICS

When you have written the C++ program for your MQSeries application, you have to create an executable application by compiling, pre-linking, and link-editing it.

MQSeries C++ for OS/390 is implemented as OS/390 DLLs for the IBM C++ for OS/390 language. Using DLLs, you have to concatenate the supplied definition side-decks with the compiler output at pre-link time. This allows the linker to check your calls to the MQSeries C++ member functions.

Note: There are three sets of side-decks for each of the three environments.

To build an MQSeries for OS/390 C++ application, you need to create and run JCL. Use the following procedure:

1. If your application runs under CICS, use the CICS-supplied procedure to translate CICS commands in your program.
2. Compile the program to produce object code. The JCL for your compilation must include statements that make the product data definition files available to the compiler. The data definitions are supplied in the following MQSeries for OS/390 libraries:
   - **thlqual**.SCSQC370
   - **thlqual**.SCSQHPPS

OS/390 applications

Be sure to specify the `/cxx` compiler option.

Note: The name **thlqual** is the high level qualifier of the MQSeries installation library on OS/390.

3. Prelink the object code created in step 1, including the following definition side-decks, which are supplied in **thlqual**.SCSQDEFS:
   - imqs23dm and imqb23dm for batch
   - imqs23dr and imqb23dr for RRS batch
   - imqs23dc and imqb23dc for CICS
4. Link-edit the object code created in step 2, to produce a load module, and store it in your application load library.

To run batch or RRS batch programs, the libraries **thlqual**.SCSQAUTH and **thlqual**.SCSQLOAD must be included in the STEPLIB, or JOBLIB data set concatenation.

To run a CICS program, you must first get your system administrator to define it to CICS as an MQSeries program and transaction. You can then run it in the usual way.

Running sample programs on OS/390

MQSeries for OS/390 supplies three sample programs, together with JCL to run them. The programs are described in "Sample programs" on page 14.

The sample applications are supplied in source form only. The files are:

Table 5. OS/390 sample program files

Sample	**Source program (**in library **thlqual.**SCSQCPPS**)**	**JCL (**in library **thlqual.**SCSQPROC**)**
HELLO WORLD	imqwrld	imqwrldr
SPUT	imqsput	imqsputr
SGET	imqsget	imqsgetr

To run the samples, you need to compile and link-edit them as with any C++ program (see "Building an application on OS/390" on page 119). Use the supplied JCL to construct and run a batch job. You must initially customize the JCL, by following the commentary included with it.

Appendix B. MQI cross-reference

This appendix contains information relating C++ to the MQI and should be read together with the *MQSeries Application Programming Reference* book.

The information covers:
- "Data structure, class, and include-file cross-reference"
- "Class attribute cross-reference" on page 122

Data structure, class, and include-file cross-reference

Table 6. Data structure, class, and include-file cross-reference

Data structure	**Class**	**Include file**
	ImqBinary	imqbin.hpp
	ImqCache	imqcac.hpp
MQCIH	ImqCICSBridgeHeader	imqcih.hpp
MQDLH	ImqDeadLetterHeader	imqdlh.hpp
MQOR	ImqDistributionList	imqdst.hpp
	ImqError	imqerr.hpp
MQGMO	ImqGetMessageOptions	imqgmo.hpp
	ImqHeader	imqhdr.hpp
MQIIH	ImqIMSBridgeHeader	imqiih.hpp
	ImqItem	imqitm.hpp
MQMD	ImqMessage	imqmsg.hpp
	ImqMessageTracker	imqmtr.hpp
	ImqNamelist	imqnml.hpp
MQOD, MQRR	ImqObject	imqobj.hpp
MQPMO, MQPMR, MQRR	ImqPutMessageOptions	imqpmo.hpp
	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		
MQWIH	ImqWorkHeader	imqwih.hpp

Class attribute cross-reference

Table 7 through 25 contain cross-reference information for each C++ class. These cross-references relate to the use of the underlying MQSeries procedural interfaces. Again, this should be read together with the *MQSeries Application Programming Reference* book. The classes ImqBinary, ImqDistributionList, and ImqString have no attributes that fall into this category and are, therefore, excluded.

ImqCache

Table 7. ImqCache cross-reference

Attribute	Call
automatic buffer	MQGET
buffer length	MQGET
buffer pointer	MQGET, MQPUT
data length	MQGET
data offset	MQGET
data pointer	MQGET
message length	MQGET, MQPUT

ImqCICSBridgeHeader

Table 8. ImqCICSBridgeHeader cross reference

Attribute	Data structure	Field
bridge abend code	MQCIH	AbendCode
ADS descriptor	MQCIH	AdsDescriptor
attention identifier	MQCIH	AttentionId
authenticator	MQCIH	Authenticator
bridge completion code	MQCIH	BridgeCompletionCode
bridge error offset	MQCIH	ErrorOffset
bridge reason code	MQCIH	BridgeReason
bridge cancel code	MQCIH	CancelCode
conversational task	MQCIH	ConversationalTask
cursor position	MQCIH	CursorPosition
facility token	MQCIH	Facility
facility keep time	MQCIH	FacilityKeepTime
facility like	MQCIH	FacilityLike
function	MQCIH	Function
get wait interval	MQCIH	GetWaitInterval
link type	MQCIH	LinkType
next transaction identifier	MQCIH	NextTransactionId
output data length	MQCIH	OutputDataLength
reply-to format	MQCIH	ReplyToFormat
bridge return code	MQCIH	ReturnCode
start code	MQCIH	StartCode

Table 8. ImqCICSBridgeHeader cross reference (continued)

Attribute	Data structure	Field
task end status	MQCIH	TaskEndStatus
transaction identifier	MQCIH	TransactionId
uow control	MQCIH	UowControl
version	MQCIH	Version

ImqDeadLetterHeader

Table 9. ImqDeadLetterHeader cross reference

Attribute	Data structure	Field
dead-letter reason code	MQDLH	Reason
destination queue manager name	MQDLH	DestQMgrName
destination queue name	MQDLH	DestQName
put application name	MQDLH	PutApplName
put application type	MQDLH	PutApplType
put date	MQDLH	PutDate
put time	MQDLH	PutTime

ImqError

Table 10. ImqError cross reference

Attribute	Call
completion code	MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET
reason code	MQBACK, MQBEGIN, MQCLOSE, MQCMIT, MQCONN, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQSET

ImqGetMessageOptions

Table 11. ImqGetMessageOptions cross reference

Attribute	Data structure	Field
group status	MQGMO	GroupStatus
match options	MQGMO	MatchOptions
message token	MQGMO	MessageToken
options	MQGMO	Options
resolved queue name	MQGMO	ResolvedQName
returned length	MQGMO	ReturnedLength
segmentation	MQGMO	Segmentation
segment status	MQGMO	SegmentStatus
	MQGMO	Signal1
	MQGMO	Signal2
syncpoint participation	MQGMO	Options
wait interval	MQGMO	WaitInterval

ImqHeader

Table 12. ImqHeader cross reference

Attribute	Data structure	Field
character set	MQDLH, MQIIH	CodedCharSetId
encoding	MQDLH, MQIIH	Encoding
format	MQDLH, MQIIH	Format
header flags	MQIIH, MQRMH	Flags

ImqIMSBridgeHeader

Table 13. ImqIMSBridgeHeader cross reference

Attribute	Data structure	Field
authenticator	MQIIH	Authenticator
commit mode	MQIIH	CommitMode
logical terminal override	MQIIH	LTermOverride
message format services map name	MQIIH	MFSMapName
reply-to format	MQIIH	ReplyToFormat
security scope	MQIIH	SecurityScope
transaction instance id	MQIIH	TranInstanceId
transaction state	MQIIH	TranState

ImqItem

Table 14. ImqItem cross reference

Attribute	Call
structure id	MQGET

ImqMessage

Table 15. ImqMessage cross reference

Attribute	Data structure	Field	Call
application id data	MQMD	ApplIdentityData	
application origin data	MQMD	ApplOriginData	
backout count	MQMD	BackoutCount	
character set	MQMD	CodedCharSetId	
encoding	MQMD	Encoding	
expiry	MQMD	Expiry	
format	MQMD	Format	
message flags	MQMD	MsgFlags	
message type	MQMD	MsgType	
offset	MQMD	Offset	

Table 15. ImqMessage cross reference (continued)

Attribute	Data structure	Field	Call
original length	MQMD	OriginalLength	
persistence	MQMD	Persistence	
priority	MQMD	Priority	
put application name	MQMD	PutApplName	
put application type	MQMD	PutApplType	
put date	MQMD	PutDate	
put time	MQMD	PutTime	
reply-to queue manager name	MQMD	ReplyToQMgr	
reply-to queue name	MQMD	ReplyToQ	
report	MQMD	Report	
sequence number	MQMD	MsgSeqNumber	
total message length		DataLength	MQGET
user id	MQMD	UserIdentifier	

ImqMessageTracker

Table 16. ImqMessageTracker cross reference

Attribute	Data structure	Field
accounting token	MQMD	AccountingToken
correlation id	MQMD	CorrelId
feedback	MQMD	Feedback
group id	MQMD	GroupId
message id	MQMD	MsgId

ImqNamelist

Table 17. ImqNamelist cross reference

Attribute	Inquiry	Call
name count	MQIA_NAME_COUNT	MQINQ
namelist name	MQCA_NAMELIST_NAME	MQINQ

ImqObject

Table 18. ImqObject cross reference

Attribute	Data structure	Field	Inquiry	Call
alteration date			MQCA_ALTERATION_DATE	MQINQ
alteration time			MQCA_ALTERATION_TIME	MQINQ
alternate user id	MQOD	AlternateUserId		
alternate security id				
close options				MQCLOSE

Table 18. ImqObject cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
description			MQCA_Q_DESC, MQCA_Q_MGR_DESC, MQCA_PROCESS_DESC	MQINQ
name	MQOD	ObjectName	MQCA_Q_MGR_NAME, MQCQ_Q_NAME, MQCA_PROCESS_NAME	MQINQ
open options				MQOPEN
open status				MQOPEN, MQCLOSE
queue manager identifier	queue manager idenitifier		MQCA_Q_MGR_IDENTIFIER	MQINQ

ImqProcess

Table 19. ImqProcess cross reference

Attribute	Inquiry	Call
application id	MQCA_APPL_ID	MQINQ
application type	MQIA_APPL_TYPE	MQINQ
environment data	MQCA_ENV_DATA	MQINQ
user data	MQCA_USER_DATA	MQINQ

ImqPutMessageOptions

Table 20. ImqPutMessageOptions cross reference

Attribute	Data structure	Field
context reference	MQPMO	Context
	MQPMO	InvalidDestCount
	MQPMO	KnownDestCount
options	MQPMO	Options
record fields	MQPMO	PutMsgRecFields
resolved queue manager name	MQPMO	ResolvedQMgrName
resolved queue name	MQPMO	ResolvedQName
	MQPMO	Timeout
	MQPMO	UnknownDestCount
syncpoint participation	MQPMO	Options

ImqQueue

Table 21. ImqQueue cross reference

Attribute	Data structure	Field	Inquiry	Call
backout requeue name			MQCA_BACKOUT_REQ_Q_NAME	MQINQ

Table 21. ImqQueue cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
backout threshold			MQIA_BACKOUT_THRESHOLD	MQINQ
base queue name			MQCA_BASE_Q_NAME	MQINQ
cluster name			MQCA_CLUSTER_NAME	MQINQ
cluster namelist name			MQCA_CLUSTER_NAMELIST	MQINQ
creation date			MQCA_CREATION_DATE	MQINQ
creation time			MQCA_CREATION_TIME	MQINQ
current depth			MQIA_CURRENT_Q_DEPTH	MQINQ
default bind			MQIA_DEF_BIND	MQINQ
default input open option			MQIA_DEF_INPUT_OPEN_OPTION	MQINQ
default persistence			MQIA_DEF_PERSISTENCE	MQINQ
default priority			MQIA_DEF_PRIORITY	MQINQ
definition type			MQIA_DEFINITION_TYPE	MQINQ
depth high event			MQIA_Q_DEPTH_HIGH_EVENT	MQINQ
depth high limit			MQIA_Q_DEPTH_HIGH_LIMIT	MQINQ
depth low event			MQIA_Q_DEPTH_LOW_EVENT	MQINQ
depth low limit			MQIA_Q_DEPTH_LOW_LIMIT	MQINQ
depth maximum event			MQIA_Q_DEPTH_MAX_LIMIT	MQINQ
distribution lists			MQIA_DIST_LISTS	MQINQ, MQSET
dynamic queue name	MQOD	DynamicQName		
harden get backout			MQIA_HARDEN_GET_BACKOUT	MQINQ
index type			MQIA_INDEX_TYPE	MQINQ
inhibit get			MQIA_INHIBIT_GET	MQINQ, MQSET
inhibit put			MQIA_INHIBIT_PUT	MQINQ, MQSET
initiation queue name			MQCA_INITIATION_Q_NAME	MQINQ
maximum depth			MQIA_MAX_Q_DEPTH	MQINQ
maximum message length			MQIA_MAX_MSG_LENGTH	MQINQ
message delivery sequence			MQIA_MSG_DELIVERY_SEQUENCE	MQINQ
next distributed queue				
open input count			MQIA_OPEN_INPUT_COUNT	MQINQ
open output count			MQIA_OPEN_OUTPUT_COUNT	MQINQ
previous distributed queue				
process name			MQCA_PROCESS_NAME	MQINQ
queue manager name	MQOD	ObjectQMgrName		
queue type			MQIA_Q_TYPE	MQINQ

Table 21. ImqQueue cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
remote queue manager name			MQCA_REMOTE_Q_MGR_NAME	MQINQ
remote queue name			MQCA_REMOTE_Q_NAME	MQINQ
resolved queue manager name	MQOD	ResolvedQMgrName		
resolved queue name	MQOD	ResolvedQName		
retention interval			MQIA_RETENTION_INTERVAL	MQINQ
scope			MQIA_SCOPE	MQINQ
service interval			MQIA_Q_SERVICE_INTERVAL	MQINQ
service interval event			MQIA_Q_SERVICE_INTERVAL_EVENT	MQINQ
shareability			MQIA_SHAREABILITY	MQINQ
storage class			MQCA_STORAGE_CLASS	MQINQ
transmission queue name			MQCA_XMIT_Q_NAME	MQINQ
trigger control			MQIA_TRIGGER_CONTROL	MQINQ, MQSET
trigger data			MQCA_TRIGGER_DATA	MQINQ, MQSET
trigger depth			MQIA_TRIGGER_DEPTH	MQINQ, MQSET
trigger message priority			MQIA_TRIGGER_MSG_PRIORITY	MQINQ, MQSET
trigger type			MQIA_TRIGGER_TYPE	MQINQ, MQSET
usage			MQIA_USAGE	MQINQ

ImqQueueManager

Table 22. ImqQueueManager cross reference

Attribute	Data structure	Field	Inquiry	Call
authority event			MQIA_AUTHORITY_EVENT	MQINQ
begin options	MQBO	Options		MQBEGIN
channel auto definition			MQIA_CHANNEL_AUTO_DEF	MQINQ
channel auto definition event			MQIA_CHANNEL_AUTO_EVENT	MQIA
channel auto definition exit			MQIA_CHANNEL_AUTO_EXIT	MQIA
character set			MQIA_CODED_CHAR_SET_ID	MQINQ
cluster workload data			MQCA_CLUSTER_WORKLOAD_DATA	MQINQ
cluster workload exit			MQCA_CLUSTER_WORKLOAD_EXIT	MQINQ
cluster workload length			MQIA_CLUSTER_WORKLOAD_ LENGTH	MQINQ

Table 22. ImqQueueManager cross reference (continued)

Attribute	Data structure	Field	Inquiry	Call
command input queue name			MQCA_COMMAND_INPUT_Q_NAME	MQINQ
command level			MQIA_COMMAND_LEVEL	MQINQ
connect options	MQCNO	Options		MQCONN, MQCONNX
connection status				MQCONN, MQCONNX, MQDISC
dead-letter queue name			MQCA_DEAD_LETTER_Q_NAME	MQINQ
default transmission queue name			MQCA_DEF_XMIT_Q_NAME	MQINQ
distribution lists			MQIA_DIST_LISTS	MQINQ
inhibit event			MQIA_INHIBIT_EVENT	MQINQ
local event			MQIA_LOCAL_EVENT	MQINQ
maximum handles			MQIA_MAX_HANDLES	MQINQ
maximum message length			MQIA_MAX_MSG_LENGTH	MQINQ
maximum priority			MQIA_MAX_PRIORITY	MQINQ
maximum uncommitted messages			MQIA_MAX_UNCOMMITTED_MSGS	MQINQ
performance event			MQIA_PERFORMANCE_EVENT	MQINQ
platform			MQIA_PLATFORM	MQINQ
remote event			MQIA_REMOTE_EVENT	MQINQ
repository name			MQCA_REPOSITORY_NAME	MQINQ
repository namelist			MQCA_REPOSITORY_NAMELIST	MQINQ
start-stop event			MQIA_START_STOP_EVENT	MQINQ
syncpoint availability			MQIA_SYNCPOINT	MQINQ
trigger interval			MQIA_TRIGGER_INTERVAL	MQINQ

ImqReferenceHeader

Table 23. ImqReferenceHeader

Attribute	Data structure	Field
destination environment	MQRMH	DestEnvLength, DestEnvOffset
destination name	MQRMH	DestNameLength, DestNameOffset
instance id	MQRMH	ObjectInstanceId
logical length	MQRMH	DataLogicalLength
logical offset	MQRMH	DataLogicalOffset
logical offset 2	MQRMH	DataLogicalOffset2
reference type	MQRMH	ObjectType
source environment	MQRMH	SrcEnvLength, SrcEnvOffset

Table 23. ImqReferenceHeader (continued)

Attribute	Data structure	Field
source name	MQRMH	SrcNameLength, SrcNameOffset

ImqTrigger

Table 24. ImqTrigger cross reference

Attribute	Data structure	Field
application id	MQTM	ApplId
application type	MQTM	ApplType
environment data	MQTM	EnvData
process name	MQTM	ProcessName
queue name	MQTM	QName
trigger data	MQTM	TriggerData
user data	MQTM	UserData

ImqWorkHeader

Table 25. ImqWorkHeader cross reference

Attribute	Data structure	Field
message token	MQWIH	MessageToken
service name	MQWIH	ServiceName
service step	MQWIH	ServiceStep

Appendix C. Reason codes

The following reason codes can occur in addition to those documented for the MQSeries MQI, in the *MQSeries Application Programming Reference*.

Note: The following list is in *alphabetic* order.

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 121) 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_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.

- The correct length for an **accounting token** is MQ_ACCOUNTING_TOKEN_LENGTH.
- The correct length for an **alternate security id** is MQ_SECURITY_ID_LENGTH.
- The correct length for a **correlation id** is MQ_CORREL_ID_LENGTH.
- The correct length for a **facility token** is MQ_FACILITY_LENGTH.
- The correct length for a **group id** is MQ_GROUP_ID_LENGTH.
- The correct length for a **message id** is MQ_MSG_ID_LENGTH.
- The correct length for an **instance id** is MQ_OBJECT_INSTANCE_ID_LENGTH.
- The correct length for a **transaction instance id** is MQ_TRAN_INSTANCE_ID_LENGTH.
- The correct length for a **message token** is MQ_MSG_TOKEN_LENGTH.

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_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_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 19).

Corrective action: set the ImqObject **open options** explicitly to cover all eventualities so that implicit reopening is not required.

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_DISTRIBUTION_LIST_EMPTY (6126 or X'17EE')
An ImqDistributionList failed to open because there are no ImqQueue objects referenced.

Corrective action: establish at least one ImqQueue object in which the **distribution list reference** addresses the ImqDistributionList object, and retry.

MQRC_ENCODING_ERROR (6106 or X'17DA')
The encoding of the (next) message item needs to be MQENC_NATIVE for pasting.

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_INCONSISTENT_OPEN_OPTIONS (6127 or X'17EF')
A method failed because the object is open, and the ImqObject **open options** are inconsistent with the required operation. The object cannot be reopened implicitly because the IMQ_IMPL_OPEN flag of the ImqObject **behavior** class attribute is false.

Corrective action: open the object with appropriate ImqObject **open options** and retry.

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_NEGATIVE_LENGTH (6117 or X'17E5')
A negative length has been supplied where a zero or positive length is required.

MQRC_NEGATIVE_OFFSET (6118 or X'17E6')
A negative offset has been supplied where a zero or positive offset 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_NO_CONNECTION_REFERENCE (6109 or X'17DD')

The **connection reference** is null. A connection to an ImqQueueManager object is required.

MQRC_NOT_CONNECTED (6124 or X'17EC')

A method failed because a required connection to a queue manager was not available, and a connection cannot be established implicitly because the IMQ_IMPL_CONN flag of the ImqQueueManager **behavior** class attribute is FALSE.

Corrective action: establish a connection to a queue manager and retry.

MQRC_NOT_OPEN (6125 or X'17ED')

A method failed because an MQSeries object was not open, and opening cannot be accomplished implicitly because the IMQ_IMPL_OPEN flag of the ImqObject **behavior** class attribute is FALSE.

Corrective action: open the object and retry.

MQRC_NULL_POINTER (6108 or X'17DC')

A null pointer has been supplied where a nonnull pointer is either required or implied.

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 19) 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 19) 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 19) 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 19) 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_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_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.

MQRC_WRONG_VERSION (6128 or X'17FO')
A method failed because a version number specified or encountered is either incorrect or not supported.

For the ImqCICSBridgeHeader class, the problem is with the **version** attribute.

Corrective action: If you are specifying a version number, use one that is supported by the class. If you are receiving message data from another program, ensure that both programs are using consistent and supported version numbers.

MQRC_ZERO_LENGTH (6116 or X'17E4')
A zero length has been supplied where a positive length is either required or implied.

The following list shows reason codes in *numeric order*.

Reason code	Decimal	Hex.
MQRC_REOPEN_EXCL_INPUT_ERROR	6100	X'000017D4'
MQRC_REOPEN_INQUIRE_ERROR	6101	X'000017D5'
MQRC_REOPEN_SAVED_CONTEXT_ERR	6102	X'000017D6'
MQRC_REOPEN_TEMPORARY_Q_ERROR	6103	X'000017D7'
MQRC_ATTRIBUTE_LOCKED	6104	X'000017D8'
MQRC_CURSOR_NOT_VALID	6105	X'000017D9'
MQRC_ENCODING_ERROR	6106	X'000017DA'
MQRC_STRUC_ID_ERROR	6107	X'000017DB'
MQRC_NULL_POINTER	6108	X'000017DC'
MQRC_NO_CONNECTION_REFERENCE	6109	X'000017DD'
MQRC_NO_BUFFER	6110	X'000017DE'
MQRC_BINARY_DATA_LENGTH_ERROR	6111	X'000017DF'
MQRC_BUFFER_NOT_AUTOMATIC	6112	X'000017E0'
MQRC_INSUFFICIENT_BUFFER	6113	X'000017E1'
MQRC_INSUFFICIENT_DATA	6114	X'000017E2'
MQRC_DATA_TRUNCATED	6115	X'000017E3'
MQRC_ZERO_LENGTH	6116	X'000017E4'
MQRC_NEGATIVE_LENGTH	6117	X'000017E5'
MQRC_NEGATIVE_OFFSET	6118	X'000017E6'
MQRC_INCONSISTENT_FORMAT	6119	X'000017E7'
MQRC_INCONSISTENT_OBJECT_STATE	6120	X'000017E8'
MQRC_CONTEXT_OBJECT_NOT_VALID	6121	X'000017E9'
MQRC_CONTEXT_OPEN_ERROR	6122	X'000017EA'
MQRC_STRUC_LENGTH_ERROR	6123	X'000017EB'
MQRC_NOT_CONNECTED	6124	X'000017EC'
MQRC_NOT_OPEN	6125	X'000017ED'
MQRC_DISTRIBUTION_LIST_EMPTY	6126	X'000017EE'
MQRC_INCONSISTENT_OPEN_OPTIONS	6127	X'000017EF'
MQRC_WRONG_VERSION	6128	X'000017F0'

Appendix D. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Programming interface information

This book is intended to help you to write application programs that run under MQSeries C++.

This book also documents General-use Programming Interface and Associated Guidance Information provided by MQSeries for C++.

General-use programming interfaces allow the customer to write programs that obtain the services of these products.

Trademarks

Changes

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.

| **Booch methodology.** An object-oriented methodology that helps users design systems using the object-oriented paradigm.

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.

cluster. A network of queue managers that are logically associated in some way.

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.

hardened message. A message that is written to auxiliary (disk) storage so that the message will not be lost in the event of a system failure.

I

ILE. Integrated Language Environment.

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.

Integrated Language Environment (ILE). The AS/400 Integrated Language Environment. This replaces the AS/400 Original Program Model (OPM).

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.

MQAI. MQSeries Administration Interface.

MQSeries Administration Interface (MQAI). A programming interface to MQSeries.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects.

O

object. In C an object is an instance of a class.

OPM. Original Program Model.

Original Program Model (OPM). The AS/400 Original Program Model. This is no longer supported on MQSeries. It is replaced by the Integrated Language Environment (ILE).

OTMA. Open Transaction Manager Access.

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.

Bibliography

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 V5.1
- MQSeries for AS/400 V5.1
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS V2.2.1.1
- MQSeries for DIGITAL UNIX (Compaq Tru64 UNIX) V2.2.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2.0.1
- MQSeries for VSE/ESA™ V2.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated.

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
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 Queue Manager Clusters
MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

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, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

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 Format (PCF) messages, and installable services.

MQSeries Administration Interface Programming Guide and Reference
The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes "AMQ" messages issued by MQSeries, applies to these MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

For other MQSeries platforms, the messages are supplied with the system. They do not appear in softcopy manual form.

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 these MQSeries products:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for AS/400 V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:
- AIX
- HP-UX

- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java™
MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries classes for Java are supported by these MQSeries products:
- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for MVS/ESA™ V1.2
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

This book is available in softcopy only.

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 V5.1 Quick Beginnings, GC33-1867

MQSeries for AS/400
MQSeries for AS/400 V5.1 Quick Beginnings, GC34-5557
MQSeries for AS/400 V5.1 System Administration, SC34-5558
MQSeries for AS/400 V5.1 Application Programming Reference (ILE RPG), SC34-5559

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

MQSeries for Compaq (DIGITAL) OpenVMS
MQSeries for Digital OpenVMS System Management Guide, GC33-1791

MQSeries for Digital UNIX (Compaq Tru64 UNIX)
MQSeries for Digital UNIX System Management Guide, GC34-5483

MQSeries for HP-UX
MQSeries for HP-UX V5.1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp
MQSeries for OS/2 Warp V5.1 Quick Beginnings, GC33-1868

MQSeries for OS/390
MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377
MQSeries for OS/390 Version 2 Release 1 Program Directory
MQSeries for OS/390 System Management Guide, SC34-5374
MQSeries for OS/390 Messages and Codes, GC34-5375
MQSeries for OS/390 Problem Determination Guide, GC34-5376

MQSeries link for R/3
MQSeries link for R/3 Version 1.2 User's Guide, GC33-1934

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

MQSeries for Sun Solaris
MQSeries for Sun Solaris V5.1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel
MQSeries for Tandem NonStop Kernel System Management Guide, GC33-1893

MQSeries for VSE/ESA
MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365
MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows
MQSeries for Windows V2.0 User's Guide, GC33-1822
MQSeries for Windows V2.1 User's Guide, GC33-1965

MQSeries for Windows NT
MQSeries for Windows NT V5.1 Quick Beginnings, GC34-5389
MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

MQSeries LotusScript Extension,
SC34-5404

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

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:
- MQSeries for AIX V5.1
- | MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

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

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of relevant MQSeries books are supplied with these MQSeries products:
- MQSeries for AIX V5.1
- | MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

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

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

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

The MQSeries product family Web site is at:

```
http://www.ibm.com/software/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 and PDF formats.
- Download MQSeries SupportPacs.

Related publications

This section describes the documentation available for related products.

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

C++ programming
Object Oriented Application Development With VisualAge® for C++ for OS/2, ISBN 0132424479.

VisualAge for C++ for AS/400 User's Guide, SC09-2416.

OTMA User's Guide.

| *ILE C++ for AS/400 Read Me First!*,
| SC09-2716.

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To 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, to this address:

  Information Development Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  United Kingdom
- 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™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-1877-03

Spine information:

MQSeries®

MQSeries Using C++