WebSphere MQ

Application Programming Guide

Version 7.0

SC34-6939-00

WebSphere MQ

Application Programming Guide

Version 7.0

SC34-6939-00

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

First edition (April 2008)

This edition of the book applies to the following:

- IBM WebSphere MQ, Version 7.0
- IBM WebSphere MQ for z/OS, Version 7.0

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

Contents

Figures

Tables

Chapter 1. Designing applications that use WebSphere MQ

Introduction to message queuing

The WebSphere® MQ products enable programs to communicate with one another across a network of unlike components (processors, operating systems, subsystems, and communication protocols) using a consistent application programming interface.

Applications designed and written using this interface are known as *message queuing* applications, because they use the *messaging* and *queuing* style:

Messaging	Programs communicate by sending each other data in messages rather than calling each other directly.
Queuing	Messages are placed on queues in storage, allowing programs to run independently of each other, at different speeds and times, in different locations, and without having a logical connection between them.

This chapter introduces messaging and queuing concepts, under these headings:

What is message queuing?

Message queuing has been used in data processing for many years. It is most commonly used today in electronic mail. Without queuing, sending an electronic message over long distances requires every node on the route to be available for forwarding messages, and the addressees to be logged on and conscious of the fact that you are trying to send them a message. In a queuing system, messages are stored at intermediate nodes until the system is ready to forward them. At their final destination they are stored in an electronic mailbox until the addressee is ready to read them.

Even so, many complex business transactions are processed today without queuing. In a large network, the system might be maintaining many thousands of connections in a ready-to-use state. If one part of the system suffers a problem, many parts of the system become unusable.

You can think of message queuing as being electronic mail for programs. In a message queuing environment, each program from the set that makes up an application suite is designed to perform a well-defined, self-contained function in response to a specific request. To communicate with another program, a program

must put a message on a predefined queue. The other program retrieves the message from the queue, and processes the requests and information contained in the message. So message queuing is a style of program-to-program communication.

Queuing is the mechanism by which messages are held until an application is ready to process them. Queuing allows you to:

- Communicate between programs (which might each be running in different environments) without having to write the communication code.
- Select the order in which a program processes messages.
- Balance loads on a system by arranging for more than one program to service a queue when the number of messages exceeds a threshold.
- Increase the availability of your applications by arranging for an alternative system to service the queues if your primary system is unavailable.

What is a message?

In message queuing, a *message* is a collection of data sent by one program and intended for another program.

WebSphere MQ defines four types of message:

Datagram	A simple message for which no reply is expected
Request	A message for which a reply is expected
Reply	A reply to a request message
Report	A message that describes an event such as the occurrence of an error

See "Types of message" on page 17 for more information about these message types.

Message descriptor

A WebSphere MQ message consists of control information and application data.

The control information is defined in a *message descriptor* structure (MQMD) and contains such things as:

- The type of the message
- An identifier for the message
- The priority for delivery of the message

The structure and content of the application data is determined by the participating programs, not by WebSphere MQ.

Message channel agent

A message channel agent moves messages from one queue manager to another.

References are made to them in this book when dealing with report messages and you will need to consider them when designing your application. See *WebSphere MQ Intercommunications* for more information.

What is a message queue?

A *message queue*, known simply as a queue, is a named destination to which messages can be sent. Messages accumulate on queues until they are retrieved by programs that service those queues.

Queues reside in, and are managed by, a queue manager (see "What is a queue manager?"). The physical nature of a queue depends on the operating system on which the queue manager is running. A queue can either be a volatile buffer area in the memory of a computer, or a data set on a permanent storage device (such as a disk). The physical management of queues is the responsibility of the queue manager and is not made apparent to the participating application programs.

Programs access queues only through the external services of the queue manager. They can open a queue, put messages on it, get messages from it, and close the queue. They can also set, and inquire about, the attributes of queues.

What is a queue manager?

A *queue manager* is a system program that provides queuing services to applications.

It provides an application programming interface so that programs can put messages on, and get messages from, queues. A queue manager provides additional functions so that administrators can create new queues, alter the properties of existing queues, and control the operation of the queue manager.

For WebSphere MQ message queuing services to be available on a system, there must be a queue manager running. You can have more than one queue manager running on a single system (for example, to separate a test system from a *live* system). To an application, each queue manager is identified by a *connection handle* (`Hconn`).

Many different applications can use the queue manager's services at the same time and these applications can be entirely unrelated. For a program to use the services of a queue manager, it must establish a connection to that queue manager.

For applications to send messages to applications that are connected to other queue managers, the queue managers must be able to communicate among themselves. WebSphere MQ implements a *store-and-forward* protocol to ensure the safe delivery of messages between such applications.

What is a cluster?

A *cluster* is a network of queue managers that are logically associated in some way. Clustering is available to queue managers on all WebSphere MQ V7.0 platforms.

In a WebSphere MQ network using distributed queuing without clustering, every queue manager is independent. If one queue manager needs to send messages to another it must have defined a transmission queue and a channel to the remote queue manager.

If you group queue managers in a cluster, the queue managers can make the queues that they host available to every other queue manager in the cluster. Then, assuming that you have the necessary network infrastructure in place, any queue manager can send a message to any other queue manager in the same cluster without the need for explicit channel definitions, remote queue definitions, or transmission queues.

There are two different reasons for using clusters: to reduce system administration and to improve availability and workload balancing.

As soon as you establish even the smallest cluster you will benefit from simplified system administration. Queue managers that are part of a cluster need fewer definitions and so the risk of making an error in your definitions is reduced.

For details of all aspects of clustering, see WebSphere MQ Queue Manager Clusters.

What is a shared queue, a queue-sharing group, and intra-group queuing?

Shared queues, queue-sharing groups, and intra-group queuing are available only on WebSphere MQ for z/OS®.

A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a sysplex. (This is not the same as a queue being *shared* by more than one application, using the same queue manager.)

The queue managers that can access the same set of shared queues form a group called a *queue-sharing group* (QSG). They communicate with each other by means of a coupling facility (CF) that stores the shared queues. See the *WebSphere MQ for z/OS Concepts and Planning Guide* for a full discussion of queue-sharing groups.

Queue managers in a queue-sharing group can communicate using normal channels or you can use a technique called *intra-group queuing* (IGQ), which lets you perform fast message transfer without defining channels.

What is a WebSphere MQ client?

WebSphere MQ clients are independently installable components of WebSphere MQ products. A client allows you to run WebSphere MQ applications, by means of a communications protocol, to interact with one or more Message Queue Interface (MQI) servers on other platforms and to connect to their queue managers.

For full details on how to install and use WebSphere MQ client components, see *WebSphere MQ Clients*.

What is publish/subscribe?

Publish/subscribe messaging allows you to decouple the provider of information from the consumers of that information. The sending application (publisher) and receiving application (subscriber) do not need to know anything about each other for the information to be sent and received.

Main features of message queuing

The main features of applications that use message queuing techniques are:

- There are no direct connections between programs.
- Communication between programs can be time-independent.
- Work can be carried out by small, self-contained programs.
- Communication can be driven by events.
- Applications can assign a priority to a message.
- Security.
- Data integrity.
- Recovery support.

No direct connections between programs

Message queuing is a technique for indirect program-to-program communication. It can be used within any application where programs communicate with each other. Communication occurs by one program putting messages on a queue (owned by a queue manager) and another program getting the messages from the queue.

Programs can get messages that were put on a queue by other programs. The other programs can be connected to the same queue manager as the receiving program, or to another queue manager. This other queue manager might be on another system, a different computer system, or even within a different business or enterprise.

There are no physical connections between programs that communicate using message queues. A program sends messages to a queue owned by a queue manager, and another program retrieves messages from the queue (see Figure 1).

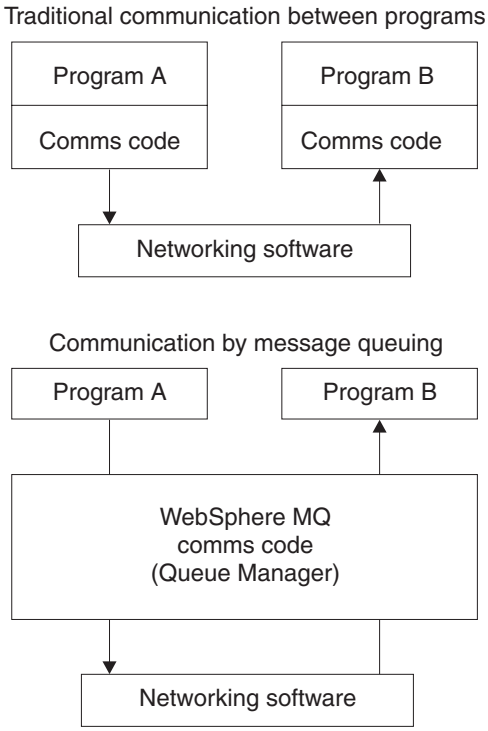


Figure 1. Message queuing compared with traditional communication

As with electronic mail, the individual messages that are part of a transaction travel through a network on a store-and-forward basis. If a link between nodes fails, the message is kept until the link is restored, or the operator or program redirects the message.

The mechanism by which a message moves from queue to queue is hidden from the programs. Therefore the programs are simpler.

Time-independent communication

Programs requesting others to do work do not have to wait for the reply to a request. They can do other work, and process the reply either when it arrives or at a later time. When writing a messaging application, you need not know (or be concerned) when a program sends a message, or when the target is able to receive the message. The message is not lost; it is retained

by the queue manager until the target is ready to process it. The message stays on the queue until it is removed by a program.

Small programs
Message queuing allows you to exploit the advantages of using small, self-contained programs. Instead of a single, large program performing all the parts of a job sequentially, you can spread the job over several smaller, independent programs. The requesting program sends messages to each of the separate programs, asking them to perform their function; when each program is complete, the results are sent back as one or more messages.

Event-driven processing
Programs can be controlled according to the state of queues. For example, you can arrange for a program to start as soon as a message arrives on a queue, or you can specify that the program does not start until there are, for example, 10 messages above a certain priority on the queue, or 10 messages of any priority on the queue.

Message priority
A program can assign a priority to a message when it puts the message on a queue. This determines the position in the queue at which the new message is added.

Programs can get messages from a queue either in the order in which the messages appear in the queue, or by getting a specific message. (A program might want to get a specific message if it is looking for the reply to a request that it sent earlier.)

Security
Authorization checks are carried out on each resource, using the tables that are set up and maintained by the WebSphere MQ administrator.

- Use Security Server (formerly known as RACF®) or other external security managers on WebSphere MQ for z/OS.
- On WebSphere MQ on UNIX® systems, Windows® systems, and i5/OS®, a security manager called the Object Authority Manager (OAM) is provided as an installable service. By default, the OAM is active.

Data integrity
Data integrity is provided by units of work. The synchronization of the start and end of units of work is fully supported as an option on each MQGET or MQPUT, allowing the results of the unit of work to be committed or rolled back. Syncpoint support operates either internally or externally to WebSphere MQ depending on the form of syncpoint coordination selected for the application.

Recovery support
For recovery to be possible, all persistent WebSphere MQ updates are logged. In the event that recovery is necessary, all persistent messages are restored, all in-flight transactions are rolled back, and any syncpoint commit and backouts are handled in the normal way of the syncpoint manager in control. For more information on persistent messages, see "Message persistence" on page 44.

WebSphere MQ clients and servers

You do not have to change a server application to support additional WebSphere MQ clients on new platforms.

Similarly, the WebSphere MQ client can, without change, function with additional types of server. See *WebSphere MQ Clients* for more information.

Benefits of message queuing to the application designer and developer

Some of the benefits of message queuing are:

- You can design applications using small programs that you can share between many applications.
- You can quickly build new applications by reusing these building blocks.
- Applications written to use message queuing techniques are not affected by changes in the way that queue managers work.
- You do not need to use any communication protocols. The queue manager deals with all aspects of communication for you.
- Programs that receive messages need not be running at the time that messages are sent to them. The messages are retained on queues.

Designers can reduce the cost of their applications because development is faster, fewer developers are needed, and demands on programming skill are lower than those for applications that do not use message queuing.

What can you do with WebSphere MQ products?

WebSphere MQ products are queue managers and application enablers. They support the IBM® Message Queue Interface (MQI) through which programs can put messages on a queue and get messages from a queue.

WebSphere MQ for z/OS

With WebSphere MQ for z/OS you can write applications that:

- Use message queuing within CICS® or IMS™.
- Send messages between batch, CICS, and IMS applications, selecting the most appropriate environment for each function.
- Send messages to applications that run on other WebSphere MQ platforms.
- Process several messages together as a single unit of work that can be committed or backed out.
- Send messages to, and interact with, IMS applications by means of the IMS bridge.
- Participate in units of work coordinated by RRS.

Each environment within z/OS has its own characteristics, advantages, and disadvantages. The advantage of WebSphere MQ for z/OS is that applications are not tied to any one environment, but can be distributed to take advantage of the benefits of each environment. For example, you can develop end-user interfaces using TSO or CICS, you can run processing-intensive modules in z/OS batch, and you can run database applications in IMS or CICS. In all cases, the various parts of the application can communicate using messages and queues.

Designers of WebSphere MQ applications must be aware of the differences and limitations imposed by these environments. For example:

- WebSphere MQ provides facilities that allow intercommunication between queue managers (this is known as *distributed queuing*).
- Methods of committing and backing out changes differ between the batch and CICS environments.

- WebSphere MQ for z/OS provides support in the IMS environment for online message processing programs (MPPs), interactive fast path programs (IFPs), and batch message processing programs (BMPs). If you are writing batch DL/I programs, follow the guidance given in this book for z/OS batch programs.
- Although multiple instances of WebSphere MQ for z/OS can exist on a single z/OS system, a CICS region can connect to only one queue manager at a time. However, more than one CICS region can be connected to the same queue manager. In the IMS and z/OS batch environments, programs can connect to more than one queue manager.
- WebSphere MQ for z/OS allows local queues to be shared by a group of queue managers, giving improved throughput and availability. Such queues are called *shared queues*, and the queue managers form a *queue-sharing group*, which can process messages on the same shared queues. Batch applications can connect to one of several queue managers within a queue-sharing group by specifying the queue-sharing group name, instead of a particular queue manager name. This is known as *group batch attach*, or more simply *group attach*. See the *WebSphere MQ for z/OS Concepts and Planning Guide* for a full discussion of queue-sharing groups.

The differences between the supported environments, and their limitations, are discussed further in "Using and writing applications on WebSphere MQ for z/OS" on page 269.

WebSphere MQ for non-z/OS platforms

With WebSphere MQ for non-z/OS platforms you can write applications that:

- Send messages to other applications running under the same operating systems. The applications can be on either the same or another system.
- Send messages to applications that run on other WebSphere MQ platforms.
- Use message queuing from within CICS for i5/OS, TXSeries® for AIX®, TXSeries for HP-UX, TXSeries for Solaris, and TXSeries for Windows systems applications.
- Use message queuing from within Encina® for AIX, HP-UX, Solaris, and Windows systems.
- Use message queuing from within Tuxedo for AIX, AT&T, HP-UX, Solaris, and Windows systems.
- Use WebSphere MQ as a transaction manager, coordinating updates made by external resource managers within WebSphere MQ units of work. The following external resource managers are supported and comply with the X/OPEN XA interface
  - DB2®
  - Informix®
  - Oracle
  - Sybase
- Process several messages together as a single unit of work that can be committed or backed out.
- Run from a full WebSphere MQ environment, or run from a WebSphere MQ client environment on the following platforms:
  - i5/OS (Java™ client only)
  - UNIX systems
  - VM/ESA®
  - Windows 2000, Windows 2003, or Windows XP

Overview of application design

This chapter introduces the design of WebSphere MQ applications, under these headings:

These subjects are discussed in greater detail in the remaining chapters of this book.

Planning your design

When you have decided how your applications can take advantage of the platforms and environments available to you, you need to decide how to use the features offered by WebSphere MQ.

Some of the key aspects are:

What types of queue should you use?
Do you want to create a queue each time that you need one, or do you want to use queues that have already been set up? Do you want to delete a queue when you have finished using it, or is it going to be used again? Do you want to use alias queues for application independence? To see what types of queues are supported, refer to "Queues" on page 50.

Should you use shared queues and queue-sharing groups (WebSphere MQ for z/OS only)?
You might want to take advantage of the increased availability, scalability, and workload balancing that are possible when you use shared queues with queue-sharing groups. See the *WebSphere MQ for z/OS Concepts and Planning Guide* for a full discussion of this topic.

Should you use queue manager clusters?
You might want to take advantage of the simplified system administration, and increased availability, scalability, and workload balancing that are possible when you use clusters. See *WebSphere MQ Queue Manager Clusters* for a full discussion of this topic.

What types of message should you use?
You might want to use datagrams for simple messages, but request messages (for which you expect replies) for other situations. You might want to assign different priorities to some of your messages.

Should you use publish/subscribe or point-to-point messaging?
Using publish/subscribe messaging, a sending application sends the information that it wants to share in a WebSphere MQ message to a standard destination managed by WebSphere MQ publish/subscribe, and lets WebSphere MQ handle the distribution of that information. The target application does not have to know anything about the source of the information it receives, it just registers an interest in one or more topics and receives that information when it is available. For more information about publish/subscribe messaging, see *WebSphere MQ Publish/Subscribe User's Guide*.

Using point-to-point messaging, a sending application sends a message to a specific queue, from where it knows a receiving application will retrieve it. A receiving application gets messages from a specific queue and acts on their contents. An application will often function both as a sender and a receiver, sending a query to another application and receiving a response.

How can you control your WebSphere MQ programs?
You might want to start some programs automatically or make programs wait until a particular message arrives on a queue (using the WebSphere MQ *triggering* feature, see "Starting WebSphere MQ applications using triggers" on page 195). Alternatively, you might want to start up another instance of an application when the messages on a queue are not getting processed fast enough (using the WebSphere MQ *instrumentation events* feature as described in *WebSphere MQ Monitoring*).

Will your application run on a WebSphere MQ client?
The full MQI is supported in the client environment and this enables almost any WebSphere MQ application to be relinked to run on a WebSphere MQ client. Link the application on the WebSphere MQ client to the MQIC library, rather than to the MQI library. Get(signal) on z/OS is not supported.

Note: An application running on a WebSphere MQ client can connect to more than one queue manager concurrently, or use a queue manager name with an asterisk (*) on an MQCONN or MQCONNX call. Change the application if you want to link to the queue manager libraries instead of the client libraries, as this function will not be available.

See *WebSphere MQ Clients* for more information.

How can you secure your data and maintain its integrity?
You can use the context information that is passed with a message to test that the message has been sent from an acceptable source. You can use the syncpointing facilities provided by WebSphere MQ or your operating system to ensure that your data remains consistent with other resources (see "Committing and backing out units of work" on page 183 for further details). You can use the *persistence* feature of WebSphere MQ messages to assure the delivery of important messages.

How should you handle exceptions and errors?
You need to consider how to process messages that cannot be delivered, and how to resolve error situations that are reported to you by the queue manager. For some reports, you must set report options on MQPUT.

The remainder of this chapter introduces the features and techniques that WebSphere MQ provides to help you answer questions like these.

Using WebSphere MQ objects

The MQI uses the following types of object:

- Queue managers
- Queues
- Administrative topic objects
- Namelists
- Services
- Listeners

- Process definitions
- Channels
- Storage classes (WebSphere MQ for z/OS only)
- Authentication information objects

These objects, and queue-sharing groups (which are only supported on WebSphere MQ for z/OS and which are not strictly objects), are discussed in "WebSphere MQ objects" on page 48.

With the exception of dynamic queues, these objects must be defined to the queue manager before you can work with them.

You define objects using:

- The PCF commands described in *WebSphere MQ Programmable Command Formats and Administration Interface*
- The MQSC commands described in *WebSphere MQ Script (MQSC) Command Reference*
- The WebSphere MQ for z/OS operations and control panels, described in the *WebSphere MQ for z/OS System Administration Guide*
- The WebSphere MQ Explorer (Windows, UNIX, and Linux® for Intel® systems only)

You can also display or alter the attributes of objects, or delete the objects.

Alternatively, for sequences of WebSphere MQ for z/OS commands that you use regularly, you can write administration programs that create messages containing commands and that put these messages on the system-command input queue. The queue manager processes the messages on this queue in the same way that it processes commands entered from the command line or from the operations and control panels. This technique is described in the *z/OS System Administration Guide*, and demonstrated in the Mail Manager sample application delivered with WebSphere MQ for z/OS. For a description of this sample, see "Sample programs for WebSphere MQ for z/OS" on page 452.

For sequences of WebSphere MQ for i5/OS commands that you use regularly you can write CL programs.

For sequences of WebSphere MQ commands on Windows systems and UNIX systems, you can use the MQSC facility to run a series of commands held in a file. For information on how to do this, see the *WebSphere MQ Script (MQSC) Command Reference*.

Designing your messages

Ask yourself these questions to help you to design the messages.

You create a message when you use an MQI call to put the message on a queue. As input to the call, you supply some control information in a *message descriptor* (MQMD) and the data that you want to send to another program. But at the design stage, you need to consider the following questions, because they affect the way that you create your messages:

What type of message should I use?

Are you designing a simple application in which you can send a message, then take no further action? Or are you asking for a reply to a question? If

you are asking a question, you might include in the message descriptor the name of the queue on which you want to receive the reply.

Do you want your request and reply messages to be synchronous? This implies that you set a timeout period for the reply to answer your request, and if you do not receive the reply within that period, it is treated as an error.

Or would you prefer to work asynchronously, so that your processes do not have to depend upon the occurrence of specific events, such as common timing signals?

Another consideration is whether you have all your messages inside a unit of work.

Should I assign different priorities to some of the messages that I create?
You can assign a priority value to each message, and define the queue so that it maintains its messages in order of their priority. If you do this, when another program retrieves a message from the queue, it always gets the message with the highest priority. If the queue does not maintain its messages in priority order, a program that retrieves messages from the queue will retrieve them in the order in which they were added to the queue.

Programs can also select a message using the identifier that the queue manager assigned when the message was put on the queue. Alternatively, you can generate your own identifiers for each of your messages.

Will my messages be discarded when the queue manager restarts?
The queue manager preserves all persistent messages, recovering them when necessary from the WebSphere MQ log files, when it is restarted. Nonpersistent messages and temporary dynamic queues are not preserved. Any messages that you do not want discarded must be defined as persistent when they are created. When writing an application for WebSphere MQ for Windows or WebSphere MQ on UNIX systems, make sure that you know how your system has been set up in respect of log file allocation to reduce the risk of designing an application that will run to the log file limits.

Because messages on shared queues (only available on WebSphere MQ for z/OS) are held in the Coupling Facility (CF), nonpersistent messages are preserved across restarts of a queue manager as long as the CF remains available. If the CF fails, nonpersistent messages are lost.

Do I want to give information about myself to the recipient of my messages?
Usually, the queue manager sets the user ID, but suitably authorized applications can also set this field, so that you can include your own user ID and other information that the receiving program can use for accounting or security purposes.

How many queues will receive my messages?
If a message might need to be put on several queues, you can use a distribution list (not on z/OS), or publish to a topic.

WebSphere MQ techniques

For a simple WebSphere MQ application, you need to decide which WebSphere MQ objects to use in your application, and which types of message you want to use. For a more advanced application, you might want to use some of the techniques introduced in the following sections.

Waiting for messages

A program that is serving a queue can await messages by:

- Waiting until either a message arrives, or a specified time interval expires (see "Waiting for messages" on page 149).
- Setting a signal so that the program is informed when a message arrives (WebSphere MQ for z/OS only). For information about this, see "Signaling" on page 150.
- Establish a call back exit to be driven when a message arrives; see "Asynchronous consumption of WebSphere MQ messages" on page 41.
- Making periodic calls on the queue to see whether a message has arrived (*polling*). This is not recommended because of the extra overhead.

Correlating replies

In WebSphere MQ applications, when a program receives a message that asks it to do some work, the program usually sends one or more reply messages to the requester.

To help the requester to associate these replies with its original request, an application can set a *correlation identifier* field in the descriptor of each message. Programs then copy the message identifier of the request message into the correlation identifier field of their reply messages.

Setting and using context information

Context information is used for associating messages with the user who generated them, and for identifying the application that generated the message. Such information is useful for security, accounting, auditing, and problem determination.

When you create a message, you can specify an option that requests that the queue manager associates default context information with your message.

For more information on using and setting context information, see "Message context" on page 46.

Starting WebSphere MQ programs automatically

Use WebSphere MQ *triggering* to start a program automatically when messages arrive on a queue.

You can set trigger conditions on a queue so that a program starts to process that queue:

- Every time that a message arrives on the queue
- When the first message arrives on the queue
- When the number of messages on the queue reaches a predefined number

For more information on triggering, see "Starting WebSphere MQ applications using triggers" on page 195. Triggering is just one way of starting a program automatically. For example, you can start a program automatically on a timer using non-WebSphere MQ facilities.

On platforms other than z/OS, WebSphere MQ can define service objects to start WebSphere MQ programs when the queue manager starts; see *WebSphere MQ Migration Information*.

Generating WebSphere MQ reports

You can request the following reports within an application:

- Exception reports
- Expiry reports
- Confirm-on-arrival (COA) reports
- Confirm-on-delivery (COD) reports
- Positive action notification (PAN) reports
- Negative action notification (NAN) reports

These are described in "Report messages" on page 19.

Clusters and message affinities

Before starting to use clusters with multiple definitions for the same queue, examine your applications to see whether there are any that require an exchange of related messages.

Within a cluster, a message can be routed to *any* queue manager that hosts an instance of the appropriate queue. Therefore, the logic of applications with message affinities can be upset.

For example, you might have two applications that rely on a series of messages flowing between them in the form of questions and answers. It might be important that all the questions are sent to the same queue manager and that all the answers are sent back to the other queue manager. In this situation, it is important that the workload management routine does not send the messages to any queue manager that just happens to host an instance of the appropriate queue.

Where possible, remove the affinities. Removing message affinities improves the availability and scalability of applications.

For more information see *WebSphere MQ Queue Manager Clusters*.

Application programming

WebSphere MQ supports the IBM Message Queue Interface (MQI). The MQI includes a set of calls with which you can send and receive messages, and manipulate WebSphere MQ objects.

Call interface

Use MQI calls to:

- Connect programs to, and disconnect programs from, a queue manager.
- Open and close objects (such as queues, queue managers, namelists, and processes).
- Put messages on queues and topics.
- Receive messages from a queue, or browse them (leaving them on the queue).
- Subscribe to topics.
- Inquire about the attributes (or properties) of WebSphere MQ objects, and set some of the attributes of queues.
- Commit and back out changes made within a unit of work, in environments where there is no natural syncpoint support, for example, UNIX systems.

- Coordinate queue manager updates and updates made by other resource managers.

The MQI provides *structures* (groups of fields) with which you supply input to, and get output from, the calls. It also provides a large set of named constants to help you supply options in the parameters of the calls. The definitions of the calls, structures, and named constants are supplied in data definition files for each of the supported programming languages. Also, default values are set within the MQI calls.

Design for performance: hints and tips

Here are a few ideas to help you to design efficient applications:

- Design your application so that processing goes on in parallel with a user's thinking time:
  - Display a panel and allow the user to start typing while the application is still initializing.
  - Get the data that you need in parallel from different servers.
- Keep connections and queues open if you are going to reuse them instead of repeatedly opening and closing, connecting and disconnecting.
- However, a server application that is putting only one message should use MQPUT1.
- Queue managers are optimized for messages between 4 KB and 100 KB in size. Very large messages are inefficient; it is probably better to send 100 messages of 1 MB each than a single 100 MB message. Very small messages are also inefficient. The queue manager does the same amount of work for a single-byte message as for a 4 KB message.
- Keep your messages within a unit of work, so that they can be committed or backed out simultaneously.
- Use the nonpersistent option for messages that do not need to be recoverable.
- If you need to send a message to a number of target queues, consider using a distribution list.

Programming platforms

For details of supported programming platforms, refer to the product announcements at

`http://www.ibm.com/software/integration/wmq/requirements`

Applications for more than one platform

Will your application run on more than one platform? Do you have a strategy to move to a different platform from the one that you use today? If the answer to either of these questions is yes, ensure that you code your programs for platform independence.

If you are using C, code in ANSI standard C. Use a standard C library function rather than an equivalent platform-specific function even if the platform-specific function is faster or more efficient. The exception is when efficiency in the code is paramount, when you should code for both situations using #ifdef. For example:

```
#ifdef _AIX
     AIX specific code
#else
     generic code
#endif
```

When you want to move the code to another platform, search the source for #ifdef with the platform specific identifiers, in this example _AIX, and add or change code as necessary.

Keep portable code in separate source files from the platform-specific code, and use a simple naming convention to split the categories.

Testing WebSphere MQ applications

The application development environment for WebSphere MQ programs is no different from that for any other application, so you can use the same development tools as well as the WebSphere MQ trace facilities.

When testing CICS applications with WebSphere MQ for z/OS, you can use the CICS Execution Diagnostic Facility (CEDF). CEDF traps the entry and exit of every MQI call as well as calls to all CICS services. Also, in the CICS environment, you can write an API-crossing exit program to provide diagnostic information before and after every MQI call. For information on how to do this, see "Using and writing applications on WebSphere MQ for z/OS" on page 269.

When testing i5/OS applications, you can use the standard Debugger. To start this, use the STRDBG command.

WebSphere MQ messages

WebSphere MQ messages are made up of two parts:

- Message properties
- Application data

Figure 2 represents a message and shows how it is logically divided into message properties and application data.

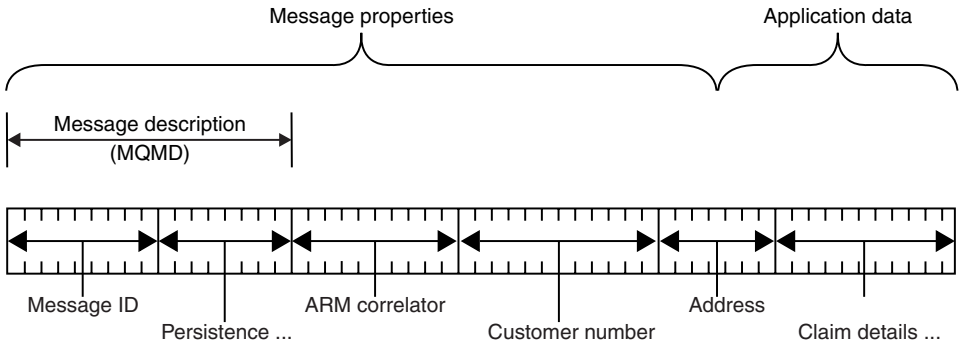


Figure 2. Representation of a message

The application data carried in a WebSphere MQ message is not changed by a queue manager unless data conversion is carried out on it. Also, WebSphere MQ does not put any restrictions on the content of this data. The length of the data in each message cannot exceed the value of the *MaxMsgLength* attribute of both the queue and queue manager.

On WebSphere MQ for AIX, WebSphere MQ for i5/OS, WebSphere MQ for HP-UX, WebSphere MQ for Linux, WebSphere MQ for Solaris, and WebSphere MQ for Windows, the *MaxMsgLength* defaults to 100 MB (104 857 600 bytes).

Note: If you are intending to use WebSphere MQ messages greater than 15 MB on i5/OS, see "Building your application on i5/OS" on page 356.

On WebSphere MQ for z/OS, the *MaxMsgLength* attribute of the queue manager is fixed at 100 MB and the *MaxMsgLength* attribute of the queue defaults to 4 MB (4 194 304 bytes) which you can change up to a maximum of 100 MB if required.

Make your messages slightly shorter than the value of the *MaxMsgLength* attribute in some circumstances. See "The data in your message" on page 113 for more information.

You create a message when you use the MQPUT or MQPUT1 MQI calls. As input to these calls, you supply the control information (such as the priority of the message and the name of a reply queue) and your data , and the call then puts the message on a queue. See the *WebSphere MQ Application Programming Reference* for more information on these calls.

This chapter introduces WebSphere MQ messages, under these headings:

Message descriptor

You can access message control information using the MQMD structure, which defines the *message descriptor*.

For a full description of the MQMD structure, see the WebSphere MQ Application Programming Reference.

See "Message context" on page 46 for a description of how to use the fields within the MQMD that contain information about the origin of the message.

There are different versions of the message descriptor. Additional information for grouping and segmenting messages (see "Message groups" on page 43) is provided in Version 2 of the message descriptor (or the MQMDE). This is the same as the Version 1 message descriptor but has additional fields. These are described in the *WebSphere MQ Application Programming Reference*.

Types of message

There are four types of message defined by WebSphere MQ:

- Datagram

- Request
- Reply
- Report

Applications can use the first three types of messages to pass information between themselves. The fourth type, report, is for applications and queue managers to use to report information about events such as the occurrence of an error.

Each type of message is identified by an MQMT_* value. You can also define your own types of message. For the range of values you can use, see the description of the `MsgType` field in the *WebSphere MQ Application Programming Reference*.

Datagrams

Use a *datagram* when you do not require a reply from the application that receives the message (that is, gets the message from the queue).

An example of an application that could use datagrams is one that displays flight information in an airport lounge. A message could contain the data for a whole screen of flight information. Such an application is unlikely to request an acknowledgement for a message because it probably does not matter if a message is not delivered. The application sends an update message after a short period of time.

Request messages

Use a *request message* when you want a reply from the application that receives the message.

An example of an application that could use request messages is one that displays the balance of a checking account. The request message could contain the number of the account, and the reply message would contain the account balance.

If you want to link your reply message with your request message, there are two options:

- Make the application that handles the request message responsible for ensuring that it puts information into the reply message that relates to the request message.
- Use the report field in the message descriptor of your request message to specify the content of the *MsgId* and *CorrelId* fields of the reply message:
  - You can request that either the *MsgId* or the `CorrelId` of the original message is to be copied into the *CorrelId* field of the reply message (the default action is to copy *MsgId*).
  - You can request that either a new `MsgId` is generated for the reply message, or that the *MsgId* of the original message is to be copied into the *MsgId* field of the reply message (the default action is to generate a new message identifier).

Reply messages

Use a *reply message* when you reply to another message.

When you create a reply message, respect any options that were set in the message descriptor of the message to which you are replying. Report options specify the content of the message identifier (`MsgId`) and correlation identifier (`CorrelId`) fields. These fields allow the application that receives the reply to correlate the reply with its original request.

Report messages

Report messages inform applications about events such as the occurrence of an error when processing a message.

They can be generated by:

- A queue manager,
- A message channel agent (for example, if they cannot deliver the message), or
- An application (for example, if it cannot use the data in the message).

Report messages can be generated at any time, and might arrive on a queue when your application is not expecting them.

Types of report message:

When you put a message on a queue, you can select to receive:

- An *exception report message*. This is sent in response to a message with the exceptions flag set. It is generated by the message channel agent (MCA) or the application.
- An *expiry report message*. This indicates that an application attempted to retrieve a message that had reached its expiry threshold; the message is marked to be discarded. This type of report is generated by the queue manager.
- A *confirmation of arrival (COA) report message*. This indicates that the message has reached its target queue. It is generated by the queue manager.
- A *confirmation of delivery (COD) report message*. This indicates that the message has been retrieved by a receiving application. It is generated by the queue manager.
- A *positive action notification (PAN) report message*. This indicates that a request has been successfully serviced (that is, the action requested in the message has been performed successfully). This type of report is generated by the application.
- A *negative action notification (NAN) report message*. This indicates that a request has *not* been successfully serviced (that is, the action requested in the message has *not* been performed successfully). This type of report is generated by the application.

Note: Each type of report message contains one of the following:

- The entire original message
- The first 100 bytes of data in the original message
- No data from the original message

You can request more than one type of report message when you put a message on a queue. If you select the delivery confirmation report message and the exception report message options, in the event that the message fails to be delivered, you receive an exception report message. However, if you select only the delivery confirmation report message option and the message fails to be delivered, you *do not* get an exception report message.

The report messages that you request, when the criteria for generating a particular message are met, are the only ones that you receive.

Report message options:

You can *discard* a message after an exception has arisen. If you select the discard option, and have requested an exception report message, the report message goes to the *ReplyToQ* and *ReplyToQMgr*, and the original message is discarded.

Note: A benefit of this is that you can reduce the number of messages going to the dead-letter queue. However, it does mean that your application, unless it sends only datagram messages, has to deal with returned messages. When an exception report message is generated, it inherits the persistence of the original message.

If a report message cannot be delivered (if the queue is full, for instance), the report message is placed on the dead-letter queue.

If you want to receive a report message, specify the name of your reply-to queue in the *ReplyToQ* field; otherwise the MQPUT or MQPUT1 of your original message fails with MQRC_MISSING_REPLY_TO_Q.

You can use other report options in the message descriptor (MQMD) of a message to specify the content of the *MsgId* and *CorrelId* fields of any report messages that are created for the message:

- You can request that either the *MsgId* or the *CorrelId* of the original message is to be copied into the *CorrelId* field of the report message. The default action is to copy the message identifier. Use MQRO_COPY_MSG_ID_TO_CORRELID because it enables the sender of a message to correlate the reply or report message with the original message. The correlation identifier of the reply or report message will be identical to the message identifier of the original message.
- You can request that either a new *MsgId* is generated for the report message, or that the *MsgId* of the original message is to be copied into the *MsgId* field of the report message. The default action is to generate a new message identifier. Use MQRO_NEW_MSG_ID because it ensures that each message in the system has a different message identifier, and can be distinguished unambiguously from all other messages in the system.
- Specialized applications might need to use MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID. However, you need to design the application that reads the messages from the queue to ensure that it works correctly when, for example, the queue contains multiple messages with the same message identifier.

  Server applications must check the settings of these flags in the request message, and set the *MsgId* and *CorrelId* fields in the reply or report message appropriately.

  Applications that act as intermediaries between a requester application and a server application should not need to check the settings of these flags. This is because these applications usually need to forward the message to the server application with the *MsgId*, *CorrelId*, and *Report* fields unchanged. This allows the server application to copy the *MsgId* from the original message in the *CorrelId* field of the reply message.

When generating a report about a message, server applications must test to see if any of these options have been set.

For more information on how to use report messages, see the description of the *Report* field in the *WebSphere MQ Application Programming Guide*.

To indicate the nature of the report, queue managers use a range of feedback codes. They put these codes in the *Feedback* field of the message descriptor of a

report message. Queue managers can also return MQI reason codes in the *Feedback* field. WebSphere MQ defines a range of feedback codes for applications to use.

For more information on feedback and reason codes, see the description of the *Feedback* field in the *WebSphere MQ Application Programming Guide*.

An example of a program that could use a feedback code is one that monitors the workloads of other programs serving a queue. If there is more than one instance of a program serving a queue, and the number of messages arriving on the queue no longer justifies this, such a program can send a report message (with the feedback code MQFB_QUIT) to one of the serving programs to indicate that the program should terminate its activity. (A monitoring program could use the MQINQ call to find out how many programs are serving a queue.)

Reports and segmented messages

Not supported on WebSphere MQ for z/OS.

If a message is segmented (see "Message segmentation" on page 144 for a description of this) and you ask for reports to be generated, you might receive more reports than you would have done had the message not been segmented.

For reports generated by WebSphere MQ:

If you segment your messages or allow the queue manager to do so, there is only one case in which you can expect to receive a single report for the entire message. This is when you have requested only COD reports, and you have specified MQGMO_COMPLETE_MSG on the getting application.

In other cases your application must be prepared to deal with several reports; usually one for each segment.

Note: If you segment your messages, and you need only the first 100 bytes of the original message data to be returned, change the setting of the report options to ask for reports with *no* data for segments that have an offset of 100 or more. If you do not do this, and you leave the setting so that each segment requests 100 bytes of data, and you retrieve the report messages with a single MQGET specifying MQGMO_COMPLETE_MSG, the reports assemble into a large message containing 100 bytes of read data at each appropriate offset. If this happens, you need a large buffer or you need to specify MQGMO_ACCEPT_TRUNCATED_MSG.

For reports generated by applications:

If your application generates reports, always copy the WebSphere MQ headers that are present at the start of the original message data to the report message data.

Then add none, 100 bytes, or all of the original message data (or whatever other amount you would usually include) to the report message data.

You can recognize the WebSphere MQ headers that must be copied by looking at the successive Format names, starting with the MQMD and continuing through any headers present. The following `Format` names indicate these WebSphere MQ headers:

- MQMDE
- MQDLH
- MQXQH

- MQIIH
- MQH*

MQH* means any name that starts with the characters MQH.

The `Format` name occurs at specific positions for MQDLH and MQXQH, but for the other WebSphere MQ headers it occurs at the same position. The length of the header is contained in a field that also occurs at the same position for MQMDE, MQIMS, and all MQH* headers.

If you are using a Version 1 MQMD, and you are reporting on a segment, or a message in a group, or a message for which segmentation is allowed, the report data must start with an MQMDE. Set the *OriginalLength* field to the length of the original message data *excluding* the lengths of any WebSphere MQ headers that you find.

Retrieval of reports:

If you ask for COA or COD reports, you can ask for them to be reassembled for you with MQGMO_COMPLETE_MSG.

An MQGET with MQGMO_COMPLETE_MSG is satisfied when enough report messages (of a single type, for example COA, and with the same *GroupId*) are present on the queue to represent one complete original message. This is true even if the report messages themselves do not contain the complete original data; the *OriginalLength* field in each report message gives the length of original data *represented* by that report message, even if the data itself is not present.

You can use this technique even if there are several different report types present on the queue (for example, both COA and COD), because an MQGET with MQGMO_COMPLETE_MSG reassembles report messages only if they have the same *Feedback* code. However, you cannot usually use this technique for exception reports, because, in general, these have different *Feedback* codes.

You can use this technique to get a positive indication that the entire message has arrived. However, in most circumstances you need to cater for the possibility that some segments arrive while others might generate an exception (or expiry, if you have allowed this). You cannot use MQGMO_COMPLETE_MSG in this case, because, in general, you might get different *Feedback* codes for different segments and, as noted above, you might get more than one report for a given segment. You can, however, use MQGMO_ALL_SEGMENTS_AVAILABLE.

To allow for this you might need to retrieve reports as they arrive, and build up a picture in your application of what happened to the original message. You can use the *GroupId* field in the report message to correlate reports with the *GroupId* of the original message, and the *Feedback* field to identify the type of each report message. The way in which you do this depends on your application requirements.

One approach is as follows:

- Ask for COD reports and exception reports.
- After a specific time, check whether a complete set of COD reports has been received using MQGMO_COMPLETE_MSG. If so, your application knows that the entire message has been processed.
- If not, and exception reports relating to this message are present, handle the problem as for unsegmented messages, but ensure that you clean up *orphan* segments at some point.

- If there are segments for which there are no reports of any kind, the original segments (or the reports) might be waiting for a channel to be reconnected, or the network might be overloaded at some point. If no exception reports at all have been received (or if you think that the ones you have might be temporary only), you might decide to let your application wait a little longer.

  As before, this is similar to the considerations you have when dealing with unsegmented messages, except that you must also consider the possibility of cleaning up *orphan* segments.

If the original message is not critical (for example, if it is a query, or a message that can be repeated later), set an expiry time to ensure that orphan segments are removed.

Back-level queue managers:

When a report is generated by a queue manager that supports segmentation, but is received on a queue manager that does *not* support segmentation, the MQMDE structure (which identifies the *Offset* and *OriginalLength* represented by the report) is always included in the report data, in addition to zero, 100 bytes, or all of the original data in the message.

However, if a segment of a message passes through a queue manager that does not support segmentation, if a report is generated there, the MQMDE structure in the original message is treated purely as data. It is not therefore included in the report data if zero bytes of the original data have been requested. Without the MQMDE, the report message might not be useful.

Request at least 100 bytes of data in reports if there is a possibility that the message might travel through a back-level queue manager.

Format of message control information and message data

The queue manager is only interested in the format of the control information within a message, whereas applications that handle the message are interested in the format of both the control information and the data.

Format of message control information

Control information in the character-string fields of the message descriptor must be in the character set used by the queue manager.

The *CodedCharSetId* attribute of the queue manager object defines this character set. Control information must be in this character set because, when applications pass messages from one queue manager to another, message channel agents that transmit the messages use the value of this attribute to determine what data conversion to perform.

Format of message data

You can specify any of the following:

- The format of the application data
- The character set of the character data
- The format of numeric data

To do this, use these fields:

Format

This indicates to the receiver of a message the format of the application data in the message.

When the queue manager creates a message, in some circumstances it uses the *Format* field to identify the format of that message. For example, when a queue manager cannot deliver a message, it puts the message on a dead-letter (undelivered message) queue. It adds a header (containing more control information) to the message, and changes the *Format* field to show this.

The queue manager has a number of *built-in formats* with names beginning MQ, for example MQFMT_STRING. If these do not meet your needs, you can define your own formats (*user-defined formats*), but you must not use names beginning with MQ for these.

When you create and use your own formats, you must write a data-conversion exit to support a program getting the message using MQGMO_CONVERT.

CodedCharSetId

This defines the character set of character data in the message. If you want to set this character set to that of the queue manager, you can set this field to the constant MQCCSI_Q_MGR or MQCCSI_INHERIT.

When you get a message from a queue, compare the value of the *CodedCharSetId* field with the value that your application is expecting. If the two values differ, you might need to convert any character data in the message or use a data-conversion message exit if one is available.

Encoding

This describes the format of numeric message data that contains binary integers, packed-decimal integers, and floating point numbers. It is usually encoded according to the particular machine on which the queue manager is running.

When you put a message on a queue, you usually specify the constant MQENC_NATIVE in the *Encoding* field. This means that the encoding of your message data is the same as that of the machine on which your application is running.

When you get a message from a queue, compare the value of the *Encoding* field in the message descriptor with the value of the constant MQENC_NATIVE on your machine. If the two values differ, you might need to convert any numeric data in the message or use a data-conversion message exit if one is available.

Application data conversion

Application data might need to be converted to the character set and the encoding required by another application where different platforms are concerned.

It can be converted at the sending queue manager, or at the receiving queue manager. If the library of built-in formats does not meet your needs, you can define your own. The type of conversion depends on the message format that is specified in the format field of the message descriptor, MQMD.

Note: Messages with MQFMT_NONE specified are not converted.

Conversion at the sending queue manager:

Set the CONVERT channel attribute to YES if you need the sending message channel agent (MCA) to convert the application data.

The conversion is performed at the sending queue manager for certain built-in formats and for user-defined formats if a suitable user exit is supplied.

Built-in formats:

These include:

- Messages that are all characters (using the format name MQFMT_STRING)
- WebSphere MQ defined messages, for example Programmable Command Formats

  WebSphere MQ uses Programmable Command Format messages for administration messages and events (the format name used is MQFMT_ADMIN in this case). You can use the same format (using the format name MQFMT_PCF) for your own messages, and take advantage of the built-in data conversion.

The queue manager built-in formats all have names beginning with MQFMT. They are listed and described in the *WebSphere MQ Application Programming Reference* under the `Format` field of the Message descriptor (MQMD).

Application-defined formats:

For user-defined formats, application data conversion must be performed by a data-conversion exit program (for more information, see "Writing data-conversion exits" on page 163). In a client-server environment, the exit is loaded at the server and conversion takes place there.

Conversion at the receiving queue manager:

Application message data can be converted by the receiving queue manager for both built-in and user-defined formats.

The conversion is performed during the processing of an MQGET call if you specify the MQGMO_CONVERT option. For details, see the *WebSphere MQ Application Programming Reference*.

Coded character sets:

WebSphere MQ products support the coded character sets that are provided by the underlying operating system.

When you create a queue manager, the queue manager coded character set ID (CCSID) used is based on that of the underlying environment. If this is a mixed code page, WebSphere MQ uses the SBCS part of the mixed code page as the queue manager CCSID.

For general data conversion, if the underlying operating system supports DBCS code pages, WebSphere MQ can use it.

See the documentation for your operating system for details of the coded character sets that it supports.

You need to consider application data conversion, format names, and user exits when writing applications that span multiple platforms. For details of the MQGET

call, the Convert characters call, the MQGMO_CONVERT option, and the built-in formats, see the *WebSphere MQ Application Programming Reference*. See "Writing data-conversion exits" on page 163 for information about invoking and writing data-conversion exits.

Message priorities

You set the priority of a message (in the *Priority* field of the MQMD structure) when you put the message on a queue. You can set a numeric value for the priority, or you can let the message take the default priority of the queue.

The *MsgDeliverySequence* attribute of the queue determines whether messages on the queue are stored in FIFO (first in, first out) sequence, or in FIFO within priority sequence. If this attribute is set to MQMDS_PRIORITY, messages are enqueued with the priority specified in the *Priority* field of their message descriptors; but if it is set to MQMDS_FIFO, messages are enqueued with the default priority of the queue. Messages of equal priority are stored on the queue in order of arrival.

The *DefPriority* attribute of a queue sets the default priority value for messages being put on that queue. This value is set when the queue is created, but it can be changed afterwards. Alias queues, and local definitions of remote queues, can have different default priorities from the base queues to which they resolve. If there is more than one queue definition in the resolution path (see "Name resolution" on page 101), the default priority is taken from the value (at the time of the put operation) of the *DefPriority* attribute of the queue specified in the open command.

The value of the *MaxPriority* attribute of the queue manager is the maximum priority that you can assign to a message processed by that queue manager. You cannot change the value of this attribute. In WebSphere MQ, the attribute has the value 9; you can create messages having priorities between 0 (the lowest) and 9 (the highest).

Message properties

Use message properties to allow an application to select messages to process, or to retrieve information about a message without accessing MQMD or MQRFH2 headers. They also facilitate communication between Websphere MQ and JMS applications.

A message property is data associated with a message, consisting of a textual name and a value of a particular type. Message properties are used by message selectors to filter publications to topics or to selectively get messages from queues. Message properties can be used to include business data or state information without having to store it in the application data. Applications do not have to access data in the MQ Message Descriptor (MQMD) or MQRFH2 headers because fields in these data structures can be accessed as message properties using Message Queue Interface (MQI) function calls.

The use of message properties in WebSphere MQ mimics the use of properties in JMS. This means that you can set properties in a JMS application and retrieve them in a procedural WebSphere MQ application, or the other way round.

Message properties and message length

Use the queue manager attribute *MaxPropertiesLength* to control the size of the properties that can flow with any message in a WebSphere MQ queue manager.

In general, when you use MQSETMP to set properties, the size of a property is the length of the property name in bytes, plus the length of the property value in bytes as passed into the MQSETMP call. It is possible for the character set of the property name and the property value to change during transmission of the message to its destination because these can be converted into Unicode; in this case the size of the property might change.

On an MQPUT or MQPUT1 call, properties of the message do not count towards the length of the message for the queue and the queue manager, but they do count towards the length of the properties as perceived by the queue manager (whether they were set using the message property MQI calls or not).

If the size of the properties exceeds the maximum properties length, the message is rejected with MQRC_PROPERTIES_TOO_BIG. Because the size of the properties is dependent on its representation, you should set the maximum properties length at a gross level.

It is possible that an application could successfully put a message with a buffer larger than the value of *MaxMsgLength* when the buffer includes properties. This is because, even when represented as MQRFH2 elements, message properties do not count towards the length of the message. The MQRFH2 header fields themselves add to the properties length only if there are one or more folders contained and every folder in the header contains properties. Otherwise the header fields count towards the message length.

On an MQGET call, properties of the message do not count towards the length of the message as far as the queue and the queue manager are concerned. However, because the properties are counted separately it is possible that the buffer returned by an MQGET call is larger than the value of the *MaxMsgLength* attribute.

Do not have your applications query the value of *MaxMsgLength* and then allocate a buffer of this size before calling MQGET; instead, allocate a buffer you consider large enough. If the MQGET fails, allocate a buffer guided by the size of the *DataLength* parameter.

The *DataLength* parameter of the MQGET call now returns the length in bytes of the application data and any properties returned in the buffer you have provided, if a message handle is not specified in the MQGMO structure.

The *Buffer* parameter of the MQPUT call now contains the application message data to be sent and any properties represented in the message data.

When flowing to a queue manager that is prior to Version 7.0 of the product, properties of the message, except those in the message descriptor, count towards the length of the message. Therefore, you should either raise the value of the *MaxMsgLength* attribute of channels going to a system prior to Version 7.0 as necessary, to compensate for the fact that more data might be sent for each message. Alternatively, you can lower the queue or queue manager *MaxMsgLength*, so that the overall level of data being sent around the system remains the same.

There is a length limit of 100 MB for message properties, excluding the message descriptor or extension for each message.

The size of a property in its internal representation is the length of the name, plus the size of its value, plus some control data for the property. There is also some control data for the set of properties after one property is added to the message.

Property names

A property name is a character string. Certain restrictions apply to its length and the set of characters that can be used.

A property name is a case-sensitive character string, limited to +4095 characters unless otherwise restricted by the context. This limit is contained in the MQ_MAX_PROPERTY_NAME_LENGTH constant.

If you exceed this maximum length when using a message property MQI call, the call fails with reason code MQRC_PROPERTY_NAME_LENGTH_ERR.

Because there is no maximum property name length in JMS, it is possible for a JMS application to set a valid JMS property name that is not a valid WebSphere MQ MQ property name when stored in an MQRFH2 structure.

In this case, when parsed, only the first 4095 characters of the property name are used; the following characters are truncated. This could cause an application using selectors to fail to match a selection string, or to match a string when not expecting to, since more than one property might truncate to the same name. When a property name is truncated, MQ issues an error log message.

All property names must follow the rules defined by the Java Language Specification for Java Identifiers, with the exception that Unicode character U+002E (".") is permitted as part of the name - but not the start. The rules for Java Identifiers equate to those contained in the JMS specification for property names.

White space characters and comparison operators are prohibited. Embedded nulls are allowed in a property name but not recommended. If you use embedded nulls, this prevents the use of the MQVS_NULL_TERMINATED constant when used with the MQCHARV structure to specify variable length strings.

Keep property names simple because applications can select messages based on the property names and the conversion between the character set of the name and of the selector might cause the selection to fail unexpectedly.

WebSphere MQ property names use character U+002E (".") for logical grouping of properties. This divides up the namespace for properties. The properties with the following prefixes, in any mixture of lower or upper case are reserved for use by the product:

- `mcd`
- `jms`
- `usr`
- `mq`
- `sib`
- `wmq`
- `Root`
- `Body`

A good way to avoid name clashes is to ensure that all applications prefix their message properties with their Internet domain name. For example, if you are developing an application using domain name "`ourcompany.com`" you could name all properties with the prefix "`com.ourcompany`". This naming convention also

allows for easy selection of properties; for example, an application can inquire on all message properties starting "`com.ourcompany.%`".

See Property name restrictions for further information about the use of property names.

Property name restrictions:

When you name a property, you must observe certain rules.

The following restrictions apply to property names:

1. A property must not begin with the following strings:
   - ″JMS″ - these are reserved for use by WebSphere MQ classes for JMS.
   - ″usr.JMS″ - these are not valid.

   The only exceptions to this are the following properties providing synonyms for JMS properties:

Property	Synonym for
JMSCorrelationID	Root.MQMD.CorrelId or jms.Cid
JMSDeliveryMode	Root.MQMD.Persistence or jms.Dlv
JMSDestination	jms.Dst
JMSExpiration	Root.MQMD.Expiry or jms.Exp
JMSMessageID	Root.MQMD.MsgId
JMSPriority	Root.MQMD.Priority or jms.Pri
JMSRedelivered	Root.MQMD.BackoutCount
JMSReplyTo (a string encoded as a URI)	Root.MQMD.ReplyToQ or Root.MQMD.ReplyToQMgr or jms.Rto
JMSTimestamp	Root.MQMD.PutDate or Root.MQMD.PutTime or jms.Tms
JMSType	mcd.Type or mcd.Set or mcd.Fmt
JMSXAppID	Root.MQMD.PutApplName
JMSXDeliveryCount	Root.MQMD.BackoutCount
JMSXGroupID	Root.MQMD.GroupId or jms.Gid
JMSXGroupSeq	Root.MQMD.MsgSeqNumber or jms.Seq
JMSXUserID	Root.MQMD.UserIdentifier

   These synonyms allow an MQI application to access JMS properties in a similar fashion to a WebSphere MQ classes for JMS client application. Of these properties, only JMSCorrelationID, JMSReplyTo, JMSType, JMSXGroupID, and JMSXGroupSeq can be set using the MQI.

   Note that the JMS_IBM_* properties available from within WebSphere MQ classes for JMS are not available using the MQI. The fields that the JMS_IBM_* properties reference can be accessed in other ways by MQI applications.
2. A property must not be called, in any mixture of lower or uppercase, "NULL", "TRUE", "FALSE", "NOT", "AND", "OR", "BETWEEN", "LIKE", "IN", "IS" and "ESCAPE". These are the names of SQL keywords used in selection strings.
3. A property beginning "mq" (except "mq_usr"), "jms", "mcd", "usr", or "sib" (in any mixture of lower or uppercase) can only contain a single "." character (U+002E).

4. Two "." characters must contain other characters in between; you cannot have an empty point in the hierarchy. Similarly a property name cannot end in a "." character.
5. If an application sets the property "a.b" and then the property "a.b.c", it is unclear whether in the hierarchy "b" contains a value or another logical grouping. Such a hierarchy is "mixed content" and this is not supported. Setting a property that causes mixed content is not allowed.

These restrictions are enforced by the validation mechanism as follows:

- Property names are validated when setting a property using the MQSETMP call, if validation was requested when the message handle was created. If an attempt to validate a property is undertaken and fails due to an error in the specification of the property name, the completion code is MQCC_FAILED with reason:
  - MQRC_PROPERTY_NAME_ERROR for reasons 1-4.
  - MQRC_MIXED_CONTENT_NOT_ALLOWED for reason 5.
- The names of properties specified directly as MQRFH2 elements are not guaranteed to be validated by the MQPUT call.

Message descriptor fields as properties:

Most message descriptor fields can be treated as properties. The property name is constructed by adding a prefix to the message descriptor field's name.

If an MQI application wants to identify a message property contained in a message descriptor field, for example, in a selector string or using the message property APIs, use the following syntax:

Property name	Message descriptor field
Root.MQMD.<Field>	<Field>

Specify `<Field>` with the same case as for the MQMD structure fields in the C language declaration. For example, the property name `Root.MQMD.AccountingToken` accesses the `AccountingToken` field of the message descriptor.

The `StrucId` and `Version` fields of the message descriptor are not accessible using the above syntax.

Message descriptor fields are never represented in an MQRFH2 header as for other properties.

If the message data starts with an MQMDE that is honored by the queue manager, the MQMDE fields can be accessed using the `Root.MQMD.<Field>` notation described above. In this case the MQMDE fields are treated as logically part of the MQMD from a properties perspective. See the section "MQMDE specified on MQPUT and MQPUT1 calls" in Overview of MQMDE.

Property data types and values

A property can be a boolean, a byte string, a character string, or a floating-point or integer number. The property can store any valid value in the range of the data type unless otherwise restricted by the context.

The data type of a property value must be one of the following values:

- MQBOOL

- MQBYTE[ ]
- MQCHAR[ ]
- MQFLOAT32
- MQFLOAT64
- MQINT8
- MQINT16
- MQINT32
- MQINT64

A property can exist but have no defined value; it is a null property. A null property is different from a byte or character string property (MQBYTE[ ] and MQCHAR[ ] respectively) that has a defined but empty value, that is, one with a zero-length value.

Byte string is not a valid property data type in JMS or XMS. You are advised not to use byte string properties in the <usr> folder.

Selecting messages from queues

You can select messages from queues using the MsgId and CorrelId fields on an MQGET call, or by using a SelectionString on an MQOPEN or MQSUB call.

Getting a specific message using *MsgId* and *CorrelId*

To get a particular message from a queue, use the *MsgId* and *CorrelId* fields of the message descriptor. If you specify Version 2 of the MQMD, you can also use the *GroupId*, *MsgSeqNumber*, and *Offset* fields.

A globally unique message identifier is ideally generated by the queue manager when a message is first put on a queue. Globally unique *MsgId*s improve serviceability because you can track messages across queue managers and locate messages in recovery logs for example. However, a WebSphere MQ application can specify a particular value for the message identifier, and although it is strongly recommended that application-generated *MsgId*s are unique identifiers, it is possible for an application to specify a non-unique *MsgId*.

You can use the correlation identifier in any way that you like. One intended use of this field is for applications to copy the message identifier of a request message into the *CorrelId* field of a reply message. Where possible use the *CorrelId* in preference to the *MsgId* if you want to associate an application-provided identity with a message. When retrieving a specific message on the distributed platforms, the queue manager is then optimized for retrieving messages by *CorrelId* (rather than by *MsgId*).

The group identifier is usually generated by the queue manager when the first message of a group is put onto a queue. The *MsgSeqNumber* field identifies the position of the message within the group and the *Offset* field identifies the segments within the message.

Where more than one message meets the combined selection criteria, the *MsgDeliverySequence* attribute of the queue determines whether messages are selected in FIFO (first in, first out) or priority order. When messages have equal priority, they are selected in FIFO order. For more information, see “The order in which messages are retrieved from a queue” on page 131.

For an example of an application that uses correlation identifiers, see "The Credit Check sample" on page 485.

Selectors

A message selector is a variable-length string used by an application to register its interest in only those messages whose properties satisfy the Structured Query Language (SQL) query that the selection string represents.

Selection using the MQSUB and MQOPEN function calls

You use the *SelectionString*, which is a structure of type MQCHARV, to make selections using the MQSUB and MQOPEN calls.

The *SelectionString* structure is used to pass a variable-length selection string to the queue manager.

The CCSID associated with the selector string is set via the VSCCSID field of the MQCHARV structure. The value used must be a CCSID that is supported for selector strings. The code pages supported are listed within the Code Page Conversion topic in the *WebSphere MQ Application Programming Reference*.

Specifying a CCSID for which there is no WebSphere MQ supported Unicode conversion, results in an error of MQRC_SOURCE_CCSID_ERROR. This error is returned at the time that the selector is presented to the queue manager, that is, on the MQSUB, MQOPEN, or MQPUT1 call.

The default value for the *VSCCSID* field is MQCCSI_APPL, which indicates that the CCSID of the selection string is equal to the queue manager CCSID, or the client CCSID if connected through a client. The MQCCSI_APPL constant can however be overridden by an application redefining it before compiling.

If the MQCHARV selector represents a NULL string, no selection takes place for that message consumer and messages are delivered as if a selector had not been used.

The maximum length of a selection string is limited only by what can be described by the MQCHARV field *VSLength*.

The SelectionString is returned on the output from an MQSUB call using the MQSO_RESUME subscribe option, if you have provided a buffer and there is a positive buffer length in VSBufSize. If you do not provide a buffer, only the length of the selection string is returned in the VSLength field of the MQCHARV. If the buffer provided is smaller than the space required to return the field, only VSBufSize bytes are returned in the provided buffer.

An application cannot alter a selection string without first closing either the handle to the queue (for MQOPEN), or subscription (for MQSUB). A new selection string can then be specified on a subsequent MQOPEN or MQSUB call.

MQOPEN
: Use MQCLOSE to close the opened handle, then specify a new selection string on a subsequent MQOPEN call.

MQSUB
: Use MQCLOSE to close the returned subscription handle (hSub), then specify a new selection string on a subsequent MQSUB call.

Figure 3 shows the process of selection using the MQSUB call.

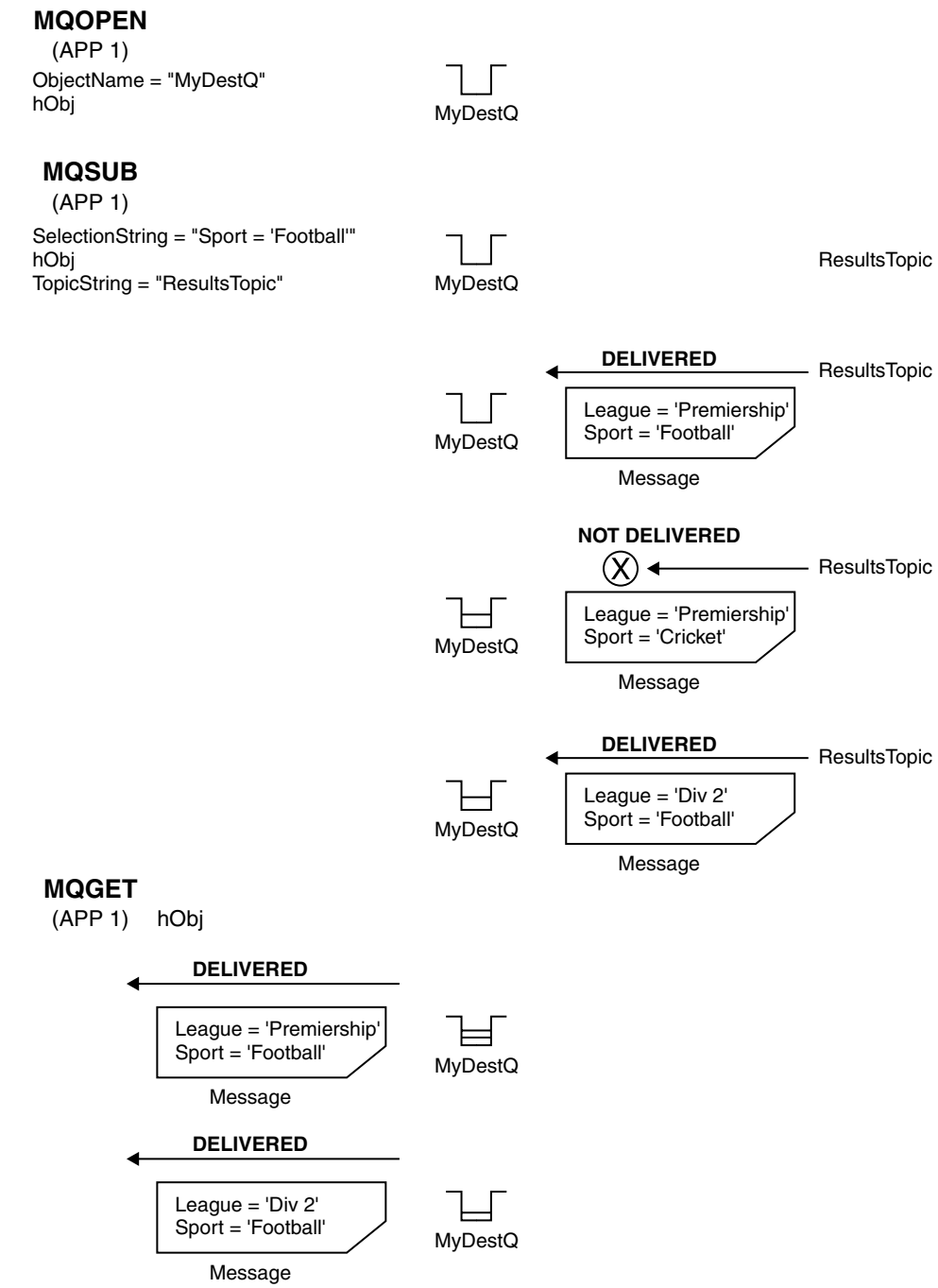


Figure 3. Selection using MQSUB call

A selector can be passed in on the call to MQSUB by using the *SelectionString* field in the MQSD structure. The effect of passing in a selector on the MQSUB is that only those messages published to the topic being subscribed to, that match a supplied selection string, are made available on the destination queue.

Figure 4 on page 34 shows the process of selection using the MQOPEN call.

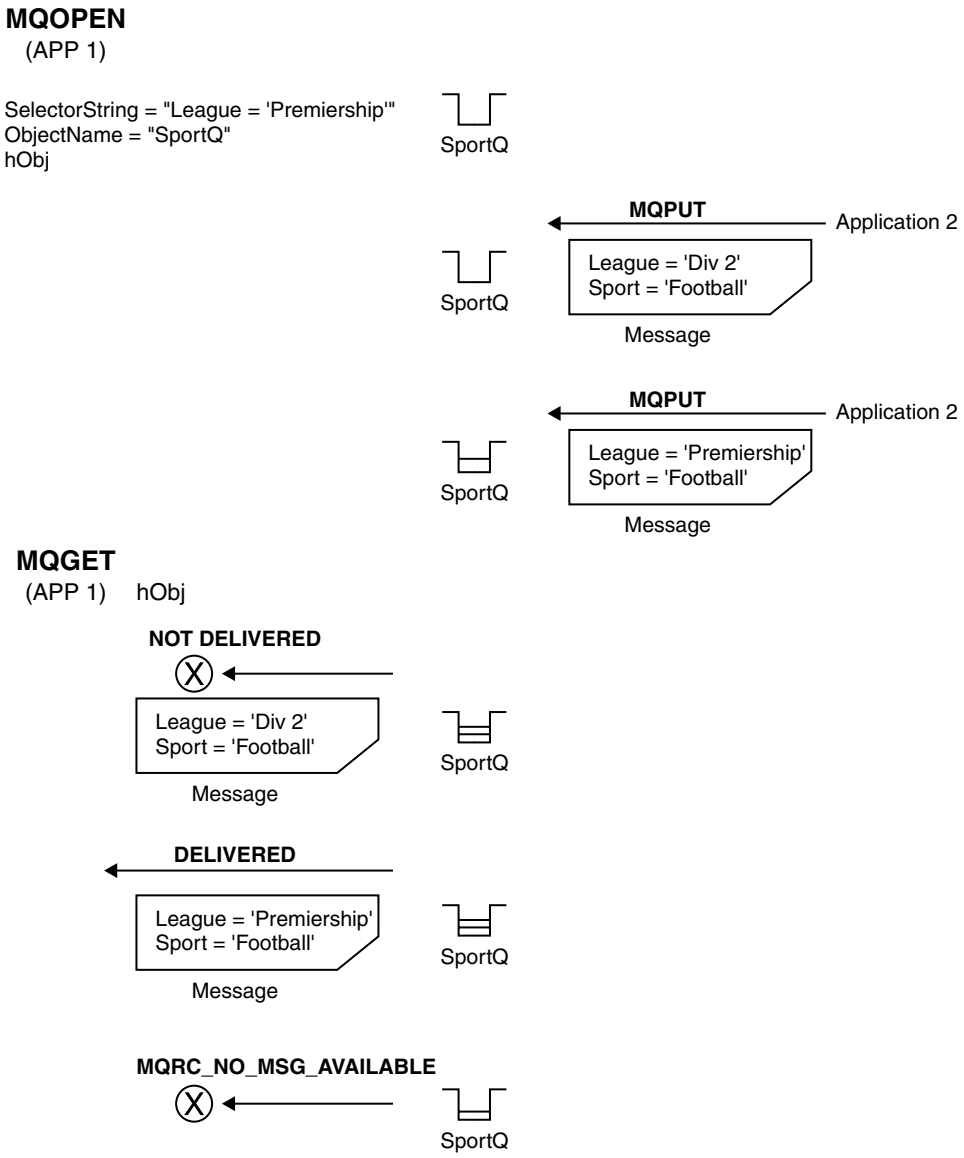


Figure 4. Selection using MQOPEN call

A selector can be passed in on the call to MQOPEN by using the *SelectionString* field in the MQOD structure. The effect of passing in a selector on the MQOPEN call is that only those messages on the opened queue, that match a selector, are delivered to the message consumer.

The main use for the selector on the MQOPEN call is for the point-to-point case where an application can elect to receive only those messages on a queue that match a selector. The example above shows a simple scenario where two messages are put to a queue opened by MQOPEN but only one is received by the application getting it, as it is the only one that matches a selector.

Note that subsequent MQGET calls result in MQRC_NO_MSG_AVAILABLE as no further messages exist on the queue that match the given selector.

Selection behavior:

Overview of WebSphere MQ selection behavior.

The fields in an MQMDE structure are considered to be the message properties for the corresponding message descriptor properties if the MQMD:

- Has format MQFMT_MD_EXTENSION
- Is immediately followed by a valid MQMDE structure
- Is version one or contains the default version two fields only

It is possible for a selection string to resolve to either TRUE or FALSE before any matching against message properties takes place. This might be the case if, for instance, the selection string is set to, for example:

```
"TRUE <>FALSE"
```

Such early evaluation is guaranteed to take place only when there are no message property references in the selection string.

If a selection string resolves to TRUE before any message properties are considered, all messages published to the topic subscribed to by the consumer are delivered. If a selection string resolves to FALSE before any message properties are considered, a reason code of MQRC_SELECTOR_ALWAYS_FALSE, and completion code MQCC_FAILED are returned on the function call that presented the selector.

Even if a message contains no message properties (other than header properties) then it may still be eligible for selection. If a selection string references a message property that does not exist, this property is assumed to have the value of NULL or 'Unknown'.

For example, a message might still satisfy a selection string like `'Color IS NULL'`, where `'Color'` does not exist as a message property in the message.

Note that selection can be performed only on the properties associated with a message, not the message itself.

Each message property has a type associated with it, and when you perform a selection, you must ensure that the values used in expressions to test message properties are of the correct type. If a type mismatch occurs, the expression in question resolves to FALSE.

It is your responsibility to ensure that the selection string and message properties use compatible types.

Selection criteria continue to be applied on behalf of inactive durable subscribers, so that only messages that match the selection string that was originally supplied are kept.

Selection strings are non-alterable when a durable subscription is resumed with alter (MQSO_ALTER). If a different selection string is presented when a durable subscriber resumes activity, then MQRC_SELECTOR_NOT_ALTERABLE is returned to the application.

Applications receive a return code of MQRC_NO_MSG_AVAILABLE if there is no message on a queue that meets the selection criteria.

If an application has specified a selection string containing property values then only those messages that contain matching properties are eligible for selection. As

an example, if a subscriber specifies a selection string of “a = 3” and a message is published containing no properties, or properties where ‘a’ does not exist or is not equal to 3, then the subscriber will not receive that message to its destination queue.

Message selector syntax:

A WebSphere MQ message selector is a String, whose syntax is based on a subset of the SQL92 conditional expression syntax.

The order in which a message selector is evaluated is from left to right within a precedence level. You can use parentheses to change this order. Predefined selector literals and operator names are written here in upper case; however, they are not case-sensitive.

WebSphere MQ verifies the syntactic correctness of a message selector at the time it is presented. If the syntax of the selection string is incorrect or a property name is not valid, a MQRC_SELECTOR_SYNTAX_ERROR is returned to the application. If property name validation was disabled when the property was set (by setting MQSMPO_NONE instead of MQSMPO_VALIDATE) and an application subsequently puts a message with in invalid property name, this message will never be selected.

A selector can contain:

- Literals:
  - String literals are enclosed in single quotes. A doubled single quote represents a single quote. Examples are ’literal’ and ’literal’’s’. Like Java string literals, these use the Unicode character encoding. You cannot use double quotes to enclose a string literal. Any sequence of bytes can be used between the quotes.
  - A byte string is one or more pair of hex characters enclosed in double quotes and prefixed by 0x. Examples are ″0x2F1C″ or ″0XD43A″. The length of a byte string must be at least one byte. When matching a selector byte string to a message property of type MQTYPE_BYTE_STRING no special action taken on leading or trailing zero, they are simply treated as another character. Endianness is also not considered. The length of both selector and property byte strings should therefore be equal and the sequence of bytes should be exactly the same.

    Examples of byte string selection (assume myBytes = 0AFC23) are:

    - “myBytes = “0x0AFC23”” = TRUE
    - “myBytes = “0xAFC23”” = MQRC_SELECTOR_SYNTAX_ERROR (because number of bytes is not multiple of two)
    - “myBytes = “0x0AFC2300”” = FALSE (because the trailing zero is significant in the comparison)
    - “myBytes = “0x000AFC23”” = FALSE (because leading zero is significant in the comparison)
    - “myBytes = “0x23FC0A”” = FALSE (because endianness is not considered)
  - Hex numbers begin with a zero, followed by an upper or lowercase ’x’. The remainder of the literal contains one or more valid hex characters. Examples are 0xA, 0xAF, 0X2020.
  - A leading zero followed by one or more digits in the range 0-7 is always interpreted as being the start of an octal number. You cannot represent a

zero-prefixed decimal number like this, for example, '09' returns a syntax error because 9 is not a valid octal digit. Examples of octal numbers are 0177, 0713.

– An exact numeric literal is a numeric value without a decimal point, such as 57, -957, and +62. An exact numeric literal can have a trailing upper or lowercase 'L' character, this does not affect how the number is stored or interpreted. WebSphere MQ supports exact numerals in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

– An approximate numeric literal is a numeric value in scientific notation, such as 7E3 or -57.9E2, or a numeric value with a decimal, such as 7., -95.7, or +6.2. WebSphere MQ supports numbers in the range -1.797693134862315E+308 to 1.797693134862315E+308.

  The significand should follow an optional sign character (+ or -). The significand should be either an integer or a fraction. A fractional part of the significand need not have a leading digit.

  An upper or lowercase 'E' indicates the start of an optional exponent. The exponent has a decimal radix and the number part of the exponent can be prefixed by an optional sign character.

  Approximate numeric literals can be terminated by an 'F' or 'D' (case insensitive). This syntax exists to support the cross-language method of tagging single or double precision numbers. These characters are optional and do not affect how an approximate numeric literal is stored or processed. These numbers are always stored and processed using double-precision.

– The boolean literals TRUE and FALSE.

Note: Non-finite IEEE-754 representations such as NaN, +Infinity, -Infinity are not supported in selection strings. It is therefore not possible to use these values as operands in an expression. Negative zero is treated the same as positive zero for mathematical operations.

- Identifiers:

  An identifier is a variable-length character sequence that must begin with a valid identifier start character, followed by zero or more valid identifier part characters. The rules for identifier names are the same as those for message property names, see "Property names" on page 28 and "Property name restrictions" on page 29 for more information.

  Identifiers are either header field references or property references. The type of a property value in a message selector must correspond to the type used to set the property, although numeric promotion is performed where possible. If a type mismatch occurs then the result of the expression is FALSE. If a property that does not exist in a message is referenced, its value is NULL.

  Type conversions that apply to the get methods for properties do not apply when a property is used in a message selector expression. For example, if you set a property as a string value and then use a selector to query it as a numeric value, the expression returns FALSE.

  JMS field and property names that map to property names or MQMD field names are also valid identifiers in a selection string. WebSphere MQ maps the recognized JMS field and property names to the message property values. See *Using Java* for more information. As an example, the selection string "JMSPriority >= "will select on the Pri property found in the jms folder of the current message.

- Overflow/underflow:

  For both decimal and approximate numeric numbers, the following are undefined:

- Specifying a number that is out of the defined range
- Specifying an arithmetic expression which would cause overflow or underflow

No checks are performed for the above conditions.

- Whitespace:

  Defined as a space, form-feed, new-line, carriage return, horizontal tab, or vertical tab. The following Unicode characters are recognized as whitespace:
  - \u0009 to \u000D
  - \u0020
  - \u001C
  - \u001D
  - \u001E
  - \u001F
  - \u1680
  - \u180E
  - \u2000 to \u200A
  - \u2028
  - \u2029
  - \u202F
  - \u205F
  - \u3000
- Expressions:
  - A selector is a conditional expression. A selector that evaluates to true matches; a selector that evaluates to false or unknown does not match.
  - Arithmetic expressions are composed of themselves, arithmetic operations, identifiers (whose value is treated as a numeric literal), and numeric literals.
  - Conditional expressions are composed of themselves, comparison operations, and logical operations.
- Standard bracketing (), to set the order in which expressions are evaluated, is supported.
- Logical operators in precedence order: NOT, AND, OR.
- Comparison operators: =, >, >=, <, <=, <> (not equal).
  - Two byte strings are equal only if the strings are of the same length and the sequence of bytes is equal.
  - Only values of the same type can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values, (the type conversion required is defined by the rules of Java numeric promotion). If there is an attempt to compare different types, the selector is always false.
  - String and boolean comparison is restricted to = and <>. Two strings are equal only if they contain the same sequence of characters.
- Arithmetic operators in precedence order:
  - +, - unary.
  - *, /, multiplication, and division.
  - +, -, addition, and subtraction.
  - Arithmetic operations on a NULL value are not supported. If they are attempted, the complete selector is always false.
  - Arithmetic operations must use Java numeric promotion.

- arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 and arithmetic-expr3 comparison operator:
  - Age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19.
  - Age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19.
  - If any of the expressions of a BETWEEN operation are NULL, the value of the operation is false. If any of the expressions of a NOT BETWEEN operation are NULL, the value of the operation is true.
- identifier [NOT] IN (string-literal1, string-literal2,...) comparison operator where identifier has a String or NULL value.
  - Country IN ('UK', 'US', 'France') is true for 'UK' and false for 'Peru'. It is equivalent to the expression (Country = 'UK') OR (Country = 'US') OR (Country = 'France').
  - Country NOT IN ('UK', 'US', 'France') is false for 'UK' and true for 'Peru'. It is equivalent to the expression NOT ((Country = 'UK') OR (Country = 'US') OR (Country = 'France')).
  - If the identifier of an IN or NOT IN operation is NULL, the value of the operation is unknown.
- identifier [NOT] LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a string value. pattern-value is a string literal, where _ stands for any single character and % stands for any sequence of characters (including the empty sequence). All other characters stand for themselves. The optional escape-character is a single character string literal, whose character is used to escape the special meaning of the _ and % in pattern-value.
  - phone LIKE '12%3' is true for 123 and 12993 and false for 1234.
  - word LIKE 'l_se' is true for lose and false for loose.
  - underscored LIKE '_%' ESCAPE '\' is true for _foo and false for bar.
  - phone NOT LIKE '12%3' is false for 123 and 12993 and true for 1234.
  - If the identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.
- identifier IS NULL comparison operator tests for a null header field value, or a missing property value.
  - prop_name IS NULL.
- identifier IS NOT NULL comparison operator tests for the existence of a non-null header field value or a property value.
  - prop_name IS NOT NULL.
- Null values

  The evaluation of selector expressions that contain NULL values is defined by SQL 92 NULL semantics, in summary:
  - SQL treats a NULL value as unknown.
  - Comparison or arithmetic with an unknown value always yields an unknown value.
  - The IS NULL and IS NOT NULL operators convert an unknown value into the respective TRUE and FALSE values.

  The boolean operators use three-valued logic (T=TRUE, F=FALSE, U=UNKNOWN)

Table 1. Boolean operator outcome when logic is A AND B

Operator A	Operator B	Outcome (A AND B)
T	F	F
T	U	U
T	T	T
F	T	F
F	U	F
F	F	F
U	T	U
U	U	U
U	F	F

Table 2. Boolean operator outcome when logic is A OR B

Operator A	Operator B	Outcome (A OR B)
T	F	T
T	U	T
T	T	T
F	T	T
F	U	U
F	F	F
U	T	T
U	U	U
U	F	U

Table 3. Boolean operator outcome when logic is NOT A

Operator A	Outcome (A AND B)
T	F
F	T
U	U

The following message selector selects messages with a message type of car, color of blue, and weight greater than 2500 lbs:

```
"JMSType = 'car' AND color = 'blue' AND weight > 250"
```

Although SQL supports fixed decimal comparison and arithmetic, message selectors do not. This is why exact numeric literals are restricted to those without a decimal. It is also why there are numerics with a decimal as an alternate representation for an approximate numeric value.

SQL comments are not supported.

Selection string rules and restrictions:

Familiarize yourself with these rules about how selection strings are interpreted and character restrictions to avoid potential problems when using selectors.

- Equivalence is tested using a single equals character, for example, “a == b” is incorrect, whereas “a = b” is correct.
- An operator used by many programming languages to represent ‘not-equals’ is ‘!=’. This representation is not a valid synonym for ‘<>’, for example, “a != b” is not valid, whereas “a <> b” is valid.
- Care must be taken to ensure that the correct type of quotes are used to contain selectors. Single quotes are recognized only if the ‘ (U+0039) character is used, not, ` (U+0145), for example is not recognized. Similarly, double quotes are valid only when used to enclose byte-strings, they are not valid for other strings.
- The symbols &, &&, | and || are not synonyms for logical conjunction/disjunction, for example. “a && b” should be specifies as “a AND b”.
- The wildcard characters * and ? are not synonyms for % and _.
- Selectors containing compound expressions such as “20 < b < 30” are not valid. Because where operators have the same precedence, the parser will evaluate from left to right, so the example would become “(20 < b) < 30”, which does not make sense. Instead the expression must be written as (b > 20) AND (b < 30).
- Byte strings must be enclosed in double quotes, if single quotes are used, the byte string will be taken to be a string literal. The number of characters (not the number which the characters represent) following the “0x” must be a multiple of two.
- The keyword ‘IS’ is not a synonym for =. Thus the selection strings “a IS 3” and “b IS ‘red’” are not valid. The ‘IS’ keyword exists only to support ‘IS NULL’ and ‘IS NOT NULL’.

Asynchronous consumption of WebSphere MQ messages

Asynchronous consumption uses a set of Message Queue Interface (MQI) extensions that allow an MQI application to be written to consume messages from a set of queues. Messages are delivered to the application by invoking a ‘unit of code’, identified by the application passing either the message, or a token representing the message.

In the most straightforward of application environments, the ‘unit of code’ is defined by a function pointer, however in other environments the ‘unit of code’ can be defined by a program or module name.

In asynchronous consumption of messages, the following terms are used:

Message consumer
A programming construct that allows you to define a program, or function, to be invoked with a message when one which matches the applications requirement becomes available.

Event handler
A programming construct that allows you to define a program or function to invoke when an asynchronous event, such as queue manager quiescing, occurs.

Call back
A generic term used to refer to either Message Consumer or an Event Handler routine.

This process provides a new programming style that can simplify the design and implementation of new applications, especially those that process multiple input queues or subscriptions.

The following illustrations give an example of how you can use this function.

Figure 5 shows a multithreaded application consuming messages from two queues. The example shows all of the messages being delivered to a single function.

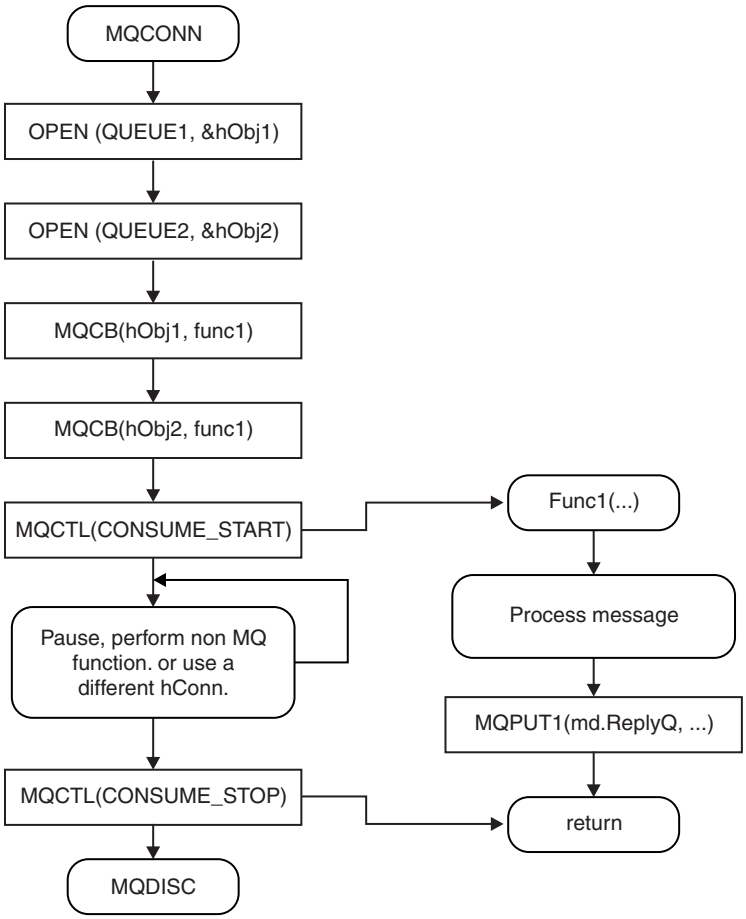


Figure 5. Standard Message Driven application consuming from two queues

Figure 6 on page 43 This sample flow shows a single threaded application consuming messages from two queues. The example shows all of the messages being delivered to a single function.

The difference from the asynchronous case is that control doesn't return to the issuer of MQCTL until all of the consumers have deactivated themselves; that is one consumer has issued an MQCTL STOP request or the queue manager quiesces.

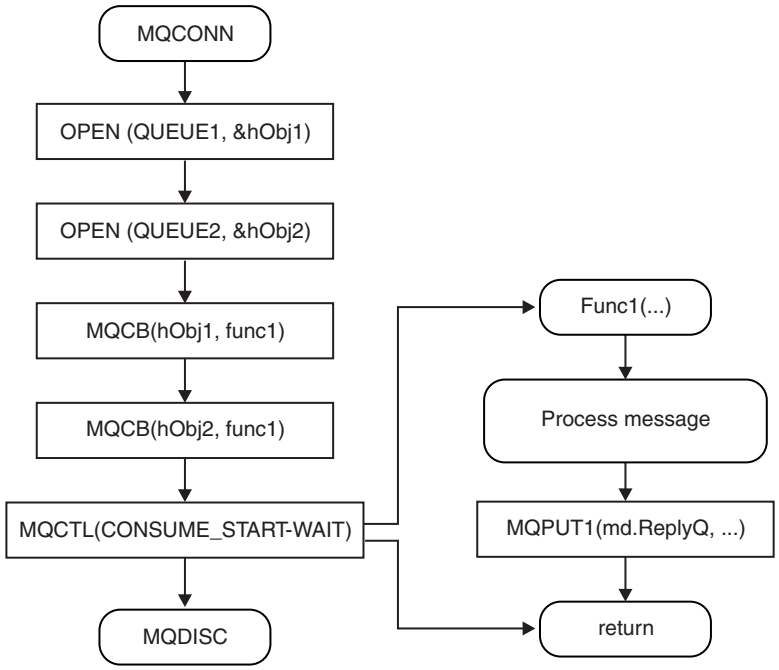


Figure 6. Single Threaded Message Driven application consuming from two queues

Message groups

Segmentation is not supported on WebSphere MQ for z/OS.

Messages can occur within groups. This allows ordering of messages (see "Logical and physical ordering" on page 131), and, except on WebSphere MQ for z/OS, segmentation of large messages (see "Message segmentation" on page 144) within the same group.

The hierarchy within a group is as follows:

Group This is the highest level in the hierarchy and is identified by a *GroupId*. It consists of one or more messages that contain the same *GroupId*. These messages can be stored anywhere on the queue.

Note: The term *message* is used here to denote one item on a queue, such as would be returned by a single MQGET that does not specify MQGMO_COMPLETE_MSG.

Figure 7 shows a group of logical messages:

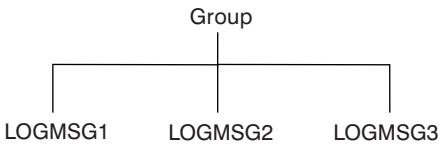


Figure 7. Group of logical messages

Logical message

Logical messages within a group are identified by the *GroupId* and *MsgSeqNumber* fields. The *MsgSeqNumber* starts at 1 for the first message within a group, and if a message is not in a group, the value of the field is 1.

Use logical messages within a group to:

- Ensure ordering (if this is not guaranteed under the circumstances in which the message is transmitted).
- Allow applications to group together similar messages (for example, those that must all be processed by the same server instance).

Each message within a group consists of one physical message, unless it is split into segments. Each message is logically a separate message, and only the *GroupId* and *MsgSeqNumber* fields in the MQMD need bear any relationship to other messages in the group. Other fields in the MQMD are independent; some might be identical for all messages in the group whereas others might be different. For example, messages in a group can have different format names, CCSIDs, encodings, and so on.

Segment

Segments are used to handle messages that are too large for either the putting or getting application or the queue manager (including intervening queue managers through which the message passes). For more information about this, see "Message segmentation" on page 144.

A segment of a message is identified by the *GroupId*, *MsgSeqNumber*, and *Offset* fields. The *Offset* field starts at zero for the first segment within a message.

Each segment consists of one physical message that might belong to a group (Figure 8 shows an example of messages within a group). A segment is logically part of a single message, so only the *MsgId*, *Offset*, and *SegmentFlag* fields in the MQMD should differ between separate segments of the same message.

Figure 8 shows a group of logical messages, some of which are segmented:

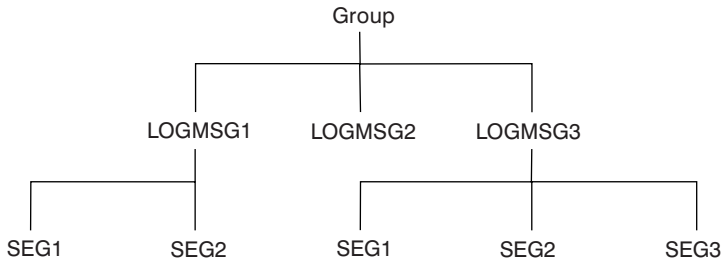


Figure 8. Segmented messages

For a description of logical and physical messages, see "Logical and physical ordering" on page 131. For further information about segmenting messages, see "Message segmentation" on page 144.

Message persistence

Persistent messages are written to logs and queue data files.

If a queue manager is restarted after a failure, it recovers these persistent messages as necessary from the logged data. Messages that are not persistent are discarded if

a queue manager stops, whether the stoppage is as a result of an operator command or because of the failure of some part of your system. Nonpersistent messages for WebSphere MQ for z/OS stored in a coupling facility (CF) are an exception to this. They persist as long as the CF remains available.

When you create a message, if you initialize the message descriptor (MQMD) using the defaults, the persistence for the message is taken from the *DefPersistence* attribute of the queue specified in the MQOPEN command. Alternatively, you can set the persistence of the message using the *Persistence* field of the MQMD structure to define the message as persistent or nonpersistent.

The performance of your application is affected when you use persistent messages; the extent of the effect depends on the performance characteristics of the machine's I/O subsystem and how you use the syncpoint options on each platform:

- A persistent message, outside the current unit of work, is written to disk on every put and get operation. See "Committing and backing out units of work" on page 183.
- In WebSphere MQ on UNIX systems, WebSphere MQ for z/OS, and WebSphere MQ for Windows, a persistent message within the current unit of work is logged only when the unit of work is committed (and the unit of work could contain many queue operations).

Nonpersistent messages can be used for fast messaging. See the *WebSphere MQ Application Programming Reference* and *WebSphere MQ Intercommunications* for further information about fast messages.

Messages that fail to be delivered

When a queue manager cannot put a message on a queue, you have various options.

You can:

- Attempt to put the message on the queue again.
- Request that the message is returned to the sender.
- Put the message on the dead-letter queue.

See "Handling program errors" on page 62 for more information.

Messages that are backed out

When processing messages from a queue under the control of a unit of work, the unit of work can consist of one or more messages. If a backout occurs, the messages that were retrieved from the queue are reinstated on the queue, and they can be processed again in another unit of work. If the processing of a particular message is causing the problem, the unit of work is backed out again. This can cause a processing loop. Messages that were put to a queue are removed from the queue.

An application can detect messages that are caught up in such a loop by testing the *BackoutCount* field of MQMD. The application can either correct the situation, or issue a warning to an operator.

In WebSphere MQ for z/OS, to ensure that the backout count for private queues survives restarts of the queue manager, set the *HardenGetBackout* attribute to MQQA_BACKOUT_HARDENED; otherwise, if the queue manager has to restart, it

does not maintain an accurate backout count for each message. Setting the attribute this way adds the penalty of extra processing.

On WebSphere MQ for i5/OS, WebSphere MQ for Windows, WebSphere MQ on UNIX systems, and shared queues on z/OS, the backout count always survives restarts of the queue manager. Any change to the *HardenGetBackout* attribute is ignored.

For more information on committing and backing out messages, see "Committing and backing out units of work" on page 183.

Reply-to queue and queue manager

There are occasions when you might receive messages in response to a message you send:

- A reply message in response to a request message
- A report message about an unexpected event or expiry
- A report message about a COA (Confirmation Of Arrival) or a COD (Confirmation Of Delivery) event
- A report message about a PAN (Positive Action Notification) or a NAN (Negative Action Notification) event

Using the MQMD structure, specify the name of the queue to which you want reply and report messages sent, in the *ReplyToQ* field. Specify the name of the queue manager that owns the reply-to queue in the *ReplyToQMgr* field.

If you leave the *ReplyToQMgr* field blank, the queue manager sets the contents of the following fields in the message descriptor on the queue:

ReplyToQ
: If *ReplyToQ* is a local definition of a remote queue, the *ReplyToQ* field is set to the name of the remote queue; otherwise this field is not changed.

ReplyToQMgr
: If *ReplyToQ* is a local definition of a remote queue, the *ReplyToQMgr* field is set to the name of the queue manager that owns the remote queue; otherwise the *ReplyToQMgr* field is set to the name of the queue manager to which your application is connected.

Note: You can request that a queue manager makes more than one attempt to deliver a message, and you can request that the message is discarded if it fails. If the message, after failing to be delivered, is not to be discarded, the remote queue manager puts the message on its dead-letter (undelivered message) queue (see "Using the dead-letter (undelivered message) queue" on page 66).

Message context

Message context information allows the application that retrieves the message to find out about the originator of the message.

The retrieving application might want to:

- Check that the sending application has the correct level of authority
- Perform some accounting function so that it can charge the sending application for any work that it has to perform
- Keep an audit trail of all the messages that it has worked with

When you use the MQPUT or MQPUT1 call to put a message on a queue, you can specify that the queue manager is to add some default context information to the message descriptor. Applications that have the appropriate level of authority can add extra context information. For more information on how to specify context information, see "Controlling context information" on page 115.

All context information is stored in the context fields of the message descriptor. The type of information falls into identity, origin, and user context information.

Identity context

Identity context information identifies the user of the application that *first* put the message on a queue:

- The queue manager fills the *UserIdentifier* field with a name that identifies the user; the way that the queue manager can do this depends on the environment in which the application is running.
- The queue manager fills the *AccountingToken* field with a token or number that it determined from the application that put the message.
- Applications can use the *ApplIdentityData* field for any extra information that they want to include about the user (for example, an encrypted password).

Suitably authorized applications can set the above fields.

A Windows systems security identifier (SID) is stored in the *AccountingToken* field when a message is created under WebSphere MQ for Windows. The SID can be used to supplement the *UserIdentifier* field and to establish the credentials of a user.

For information on how the queue manager fills the *UserIdentifier* and *AccountingToken* fields, see the descriptions of these fields in the *WebSphere MQ Application Programming Reference*.

Applications that pass messages from one queue manager to another should also pass on the identity context information so that other applications know the identity of the originator of the message.

Origin context

Origin context information describes the application that put the message on the queue on which the message is *currently* stored. The message descriptor contains the following fields for origin context information:

PutApplType	The type of application that put the message (for example, a CICS transaction).
PutApplName	The name of the application that put the message (for example, the name of a job or transaction).
PutDate	The date on which the message was put on the queue.
PutTime	The time at which the message was put on the queue.
ApplOriginData	Any extra information that an application wants to include about the origin of the message. For example, it could be set by suitably authorized applications to indicate whether the identity data is trusted.

Origin context information is usually supplied by the queue manager. Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields. See the descriptions of these fields in the *WebSphere MQ Application Programming Reference*.

An application with enough authority can provide its own context. This allows accounting information to be preserved when a single user has a different user ID on each of the systems that process a message that they have originated.

User context

User context allows an application to pass properties around a queue manager network without direct support of message handles on all the intermediate applications.

You can include any message property in the user context, by setting the Context field of the message property descriptor (MQPD) when you make the MQSETMP call.

WebSphere MQ objects

The WebSphere MQ objects are:

- Queue managers
- Queue-sharing groups (WebSphere MQ for z/OS only), although these are not strictly objects.
- Queues
- Administrative topic objects
- Namelists
- Process definitions
- Authentication information objects
- Channels
- Storage classes (WebSphere MQ for z/OS only)
- Listeners
- Services (not WebSphere MQ for z/OS)

Queue managers define the properties (known as attributes) of these objects. The values of these attributes affect the way in which WebSphere MQ processes these objects. From your applications, you use the Message Queue Interface (MQI) to control these objects. Objects are identified by an *object descriptor* (MQOD) when addressed from a program.

When you use WebSphere MQ commands to define, alter, or delete objects, for example, the queue manager checks that you have the required level of authority to perform these operations. Similarly, when an application uses the MQOPEN call to open an object, the queue manager checks that the application has the required level of authority before it allows access to that object. The checks are made on the name of the object being opened.

This chapter introduces WebSphere MQ objects, under these headings:

- "Queue managers" on page 49
- "Queue-sharing groups" on page 50
- "Queues" on page 50
- "Administrative topic objects" on page 58

Queue managers

A *queue manager* supplies an application with WebSphere MQ services.

A program must have a connection to a queue manager before it can use the services of that queue manager. A program can make this connection explicitly (using the MQCONN or MQCONNX call), or the connection might be made implicitly (this depends on the platform and the environment in which the program is running).

Queues belong to queue managers, but programs can send messages to queues that belong to any queue manager.

Attributes of queue managers

Associated with each queue manager is a set of attributes (or properties) that define its characteristics. Some of the attributes of a queue manager are fixed when it is created; you can change others using the WebSphere MQ commands. You can inquire about the values of *all* the attributes, except those used for Secure Sockets Layer (SSL) encryption, using the MQINQ call.

The *fixed* attributes include:

- The name of the queue manager
- The platform on which the queue manager runs (for example, z/OS)
- The level of system control commands that the queue manager supports
- The maximum priority that you can assign to messages processed by the queue manager
- The name of the queue to which programs can send WebSphere MQ commands
- The maximum length of messages the queue manager can process (fixed only in WebSphere MQ for z/OS)
- Whether the queue manager supports syncpointing when programs put and get messages

The *changeable* attributes include:

- A text description of the queue manager
- The identifier of the character set the queue manager uses for character strings when it processes MQI calls
- The time interval that the queue manager uses to restrict the number of trigger messages
- The time interval that the queue manager uses to determine how often queues are to be scanned for expired messages (WebSphere MQ for z/OS only)
- The name of the queue manager's dead-letter (undelivered message) queue
- The name of the queue manager's default transmission queue

- The maximum number of open handles for any one connection
- The enabling and disabling of various categories of event reporting
- The maximum number of uncommitted messages within a unit of work

For a full description of all the attributes, see the *WebSphere MQ Application Programming Reference*.

Queue managers and workload management

You can set up a cluster of queue managers that has more than one definition for the same queue (for example, the queue managers in the cluster could be clones of each other). Messages for a particular queue can be handled by any queue manager that hosts an instance of the queue. A workload-management algorithm decides which queue manager handles the message and so spreads the workload between your queue managers. See *WebSphere MQ Queue Manager Clusters* for further information.

Queue-sharing groups

Supported only on WebSphere MQ for z/OS.

Queue-sharing groups are not strictly objects, but are mentioned here for convenience.

Queue managers that can access the same set of shared queues form a group called a *queue-sharing group* (QSG), and they communicate with each other by means of a coupling facility (CF) that stores the shared queues. A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a queue-sharing group. (This is not the same as a queue being *shared* by more than one application, using the same queue manager.) See the *WebSphere MQ for z/OS Concepts and Planning Guide* for a full discussion of shared queues and queue-sharing groups.

Queues

A WebSphere MQ *queue* is a named object on which applications can put messages, and from which applications can get messages.

Messages are stored on a queue, so that if the putting application is expecting a reply to its message, it is free to do other work while waiting for that reply. Applications access a queue by using the Message Queue Interface (MQI), described in "Introducing the Message Queue Interface" on page 69.

Before a message can be put on a queue, the queue must have already been created. A queue is owned by a queue manager, and that queue manager can own many queues. However, each queue must have a name that is unique within that queue manager.

A queue is maintained through a queue manager. In most cases, each queue is physically managed by its queue manager but this is transparent to an application program. WebSphere MQ for z/OS shared queues can be managed by any queue manager in the queue-sharing group.

To create a queue you can use WebSphere MQ commands (MQSC), PCF commands, or platform-specific interfaces such as the WebSphere MQ for z/OS operations and control panels.

You can create local queues for temporary jobs *dynamically* from your application. For example, you can create *reply-to* queues (which are not needed after an application ends). For more information, see "Dynamic queues" on page 55.

Before using a queue, you must open the queue, specifying what you want to do with it. For example, you can open a queue for:

- Browsing messages only (not retrieving them)
- Retrieving messages (and either sharing the access with other programs, or with exclusive access)
- Putting messages on the queue
- Inquiring about the attributes of the queue
- Setting the attributes of the queue

For a complete list of the options that you can specify when you open a queue, see the description of the MQOPEN call in the *WebSphere MQ Application Programming Reference*.

Types of queue

The types of queue that WebSphere MQ supports for applications to use are:

Local and remote queues

A queue is known to a program as *local* if it is owned by the queue manager to which the program is connected; the queue is known as *remote* if it is owned by a different queue manager. The important difference between these two types of queue is that you can get messages only from local queues. (You can put messages on both types of queue.)

The queue definition object, created when you define a local queue, holds the definition information of the queue as well as the physical messages put on the queue. The queue definition object, created when you *define* a remote queue, only holds the information necessary for the local queue manager to locate the queue to which you want your message to go. This object is known as the *local definition of a remote queue*. All the attributes of the remote queue are held by the queue manager that owns it, because it is a local queue to that queue manager.

Shared queues

Shared queues are available only on WebSphere MQ for z/OS.

A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a queue-sharing group. (This is not the same as a queue being *shared* by more than one application, using the same queue manager.) Shared queues are held by a coupling facility (CF), and are accessible by any queue manager in the queue-sharing group. Each shared queue in a queue-sharing group must have a name that is unique within that group. See the *WebSphere MQ for z/OS Concepts and Planning Guide* for a full discussion of shared queues and queue-sharing groups.

Alias queues

To your program, an *alias queue* appears to be a queue, but it is really a WebSphere MQ object that you can use to access another queue or a topic. This means that more than one program can work with the same queue, accessing it using different names. For more information about topic aliases, see *WebSphere MQ Publish/Subscribe User's Guide*.

Model and dynamic queues
A model queue is a template of a queue definition used only when you want to create a dynamic local queue.

You can create a local queue dynamically from a WebSphere MQ program, naming the model queue that you want to use as the template for the queue attributes. At that point you can change some attributes of the new queue. However, you cannot change the *DefinitionType*. If, for example, you require a permanent queue, select a model queue with the definition type set to permanent. Some conversational applications can use dynamic queues to hold replies to their queries because they probably do not need to maintain these queues after they have processed the replies.

Cluster queues
A cluster queue is a queue that is hosted by a cluster queue manager and made available to other queue managers in the cluster.

The cluster queue manager makes a local queue definition for the queue specifying the name of the cluster that the queue is to be available in. This definition has the effect of advertising the queue to the other queue managers in the cluster. The other queue managers in the cluster can put messages to a cluster queue without needing a corresponding remote-queue definition. A cluster queue can be advertised in more than one cluster. See "What is a cluster?" on page 3 and *WebSphere MQ Queue Manager Clusters* for further information.

Types of local queue:

Each queue manager can have some local queues that it uses for special purposes:

Transmission queues
A *transmission queue* is a local queue that holds messages destined for a remote queue. The messages are forwarded to their destination queue by WebSphere MQ when a communication program and link are available.

Initiation queues
An *initiation queue* is a local queue on which the queue manager puts a message to start an application when certain conditions (such as more than 10 messages arriving, for example) are met on a local queue.

Dead-letter (undelivered message) queue
The *dead-letter queue* is a local queue on which the queue manager and applications put messages they cannot deliver. You need to process any messages that arrive on this queue.

System command queue
The *system command queue* is a queue to which suitably authorized applications can send WebSphere MQ commands.

System default queues
When you create a queue (other than a dynamic queue), WebSphere MQ uses the queue definitions stored in the *system default queues*.

Channel queues
Channel queues are used for distributed queue management.

Event queues
Event queues hold event messages. These messages are reported by the queue manager or a channel.

These special queues are described in greater detail in the following sections.

Attributes of queues

Some of the attributes of a queue are specified when the queue is defined, and cannot be changed afterwards (for example, the type of the queue). Other attributes of queues can be grouped into those that can be changed:

- By the queue manager during the processing of the queue (for example, the current depth of a queue)
- Only by commands (for example, the text description of the queue)
- By applications, using the MQSET call (for example, whether or not put operations are allowed on the queue)

You can find the values of all the attributes using the MQINQ call.

The attributes that are common to more than one type of queue are:

QName Name of the queue

QType Type of the queue

QDesc Text description of the queue

InhibitGet
: Whether or not programs are allowed to get messages from the queue (although you can never get messages from remote queues)

InhibitPut
: Whether or not programs are allowed to put messages on the queue

DefPriority
: Default priority for messages put on the queue

DefPersistence
: Default persistence for messages put on the queue

Scope (not supported on z/OS)
: Controls whether an entry for this queue also exists in a name service

For a full description of these attributes, see the *WebSphere MQ Application Programming Reference*.

Remote queues

To a program, a queue is *remote* if it is owned by a different queue manager to the one to which the program is connected.

Where a communication link has been established, a program can send a message to a remote queue. A program can never get a message from a remote queue.

When opening a remote queue, to identify the queue you must specify either:

- The name of the local definition that defines the remote queue.

  To create a local definition of a remote queue use the DEFINE QREMOTE command; on WebSphere MQ for i5/OS, use the CRTMQMQ command.

  From the viewpoint of an application, this is the same as opening a local queue. An application does not need to know if a queue is local or remote.
- The name of the remote queue manager and the name of the queue as it is known to that remote queue manager.

Local definitions of remote queues have three attributes in addition to the common attributes described in "Attributes of queues." These are *RemoteQName* (the name

that the queue's owning queue manager knows it by), *RemoteQMgrName* (the name of the owning queue manager), and *XmitQName* (the name of the local transmission queue that is used when forwarding messages to other queue managers). For a fuller description of these attributes, see the *WebSphere MQ Application Programming Reference*.

If you use the MQINQ call against the local definition of a remote queue, the queue manager returns the attributes of the local definition only, that is the remote queue name, the remote queue manager name, and the transmission queue name, not the attributes of the matching local queue in the remote system.

See also "Transmission queues" on page 57.

Alias queues

An *alias queue* is a WebSphere MQ object that you can use to access another queue or a topic.

The queue resulting from the resolution of an alias name (known as the base queue) can be a local queue, the local definition of a remote queue, or a shared queue (a type of local queue only available on WebSphere MQ for z/OS). It can also be either a predefined queue or a dynamic queue, as supported by the platform.

An alias name can also resolve to a topic. If an application currently puts messages onto a queue, it can be made to publish to a topic by making the queue name an alias for the topic. No change to the application code is necessary. For more information about topic aliases, see *WebSphere MQ Publish/Subscribe User's Guide*.

Note: An alias cannot resolve to another alias.

An example of the use of alias queues is for a system administrator to give different access authorities to the base queue name (that is, the queue to which the alias resolves) and to the alias queue name. This means that a program or user can be authorized to use the alias queue, but not the base queue.

Alternatively, authorization can be set to inhibit put operations for the alias name, but allow them for the base queue.

In some applications, the use of alias queues means that system administrators can easily change the definition of an alias queue object without having to get the application changed.

WebSphere MQ makes authorization checks against the alias name when programs try to use that name. It does not check that the program is authorized to access the name to which the alias resolves. A program can therefore be authorized to access an alias queue name, but not the resolved queue name.

In addition to the general queue attributes described in "Attributes of queues" on page 53, alias queues have a *BaseQName* attribute. This is the name of the base queue to which the alias name resolves. For a fuller description of this attribute, see the *WebSphere MQ Application Programming Reference*.

The *InhibitGet* and *InhibitPut* attributes (see "Attributes of queues" on page 53) of alias queues belong to the alias name. For example, if the alias-queue name

ALIAS1 resolves to the base-queue name BASE, inhibitions on ALIAS1 affect ALIAS1 only and BASE is not inhibited. However, inhibitions on BASE also affect ALIAS1.

The *DefPriority* and *DefPersistence* attributes also belong to the alias name. So, for example, you can assign different default priorities to different aliases of the same base queue. Also, you can change these priorities without having to change the applications that use the aliases.

Model queues

A *model queue* is a template of a queue definition that you use when creating a dynamic queue.

You specify the name of a model queue in the *object descriptor* (MQOD) of your MQOPEN call. Using the attributes of the model queue, the queue manager dynamically creates a local queue for you.

You can specify a name (in full) for the dynamic queue, or the stem of a name (for example, ABC) and let the queue manager add a unique part to this, or you can let the queue manager assign a complete unique name for you. If the queue manager assigns the name, it puts it in the MQOD structure.

You cannot issue an MQPUT1 call directly to a model queue , but you can issue an MQPUT1 to the dynamic queue that has been created by opening a model queue.

The attributes of a model queue are a subset of those of a local queue. For a fuller description, see the *WebSphere MQ Application Programming Reference*.

Dynamic queues

When an application program issues an MQOPEN call to open a model queue, the queue manager dynamically creates an instance of a local queue with the same attributes as the model queue.

Depending on the value of the *DefinitionType* field of the model queue, the queue manager creates either a temporary or permanent dynamic queue (See “Creating dynamic queues” on page 107).

Properties of temporary dynamic queues:

Temporary dynamic queues have the following properties:

- They cannot be shared queues, accessible from queue managers in a queue-sharing group (only available on WebSphere MQ for z/OS).
- They hold nonpersistent messages only.
- They are non-recoverable.
- They are deleted when the queue manager is started.
- They are deleted when the application that issued the MQOPEN call that created the queue closes the queue or terminates.
  - If there are any committed messages on the queue, they are deleted.
  - If there are any uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue at this time, the queue is marked as being logically deleted, and is only physically deleted (after these calls have been committed) as part of close processing, or when the application terminates.

- If the queue is in use at this time (by the creating, or another application), the queue is marked as being logically deleted, and is only physically deleted when closed by the last application using the queue.
- Attempts to access a logically deleted queue (other than to close it) fail with reason code MQRC_Q_DELETED.
- MQCO_NONE, MQCO_DELETE and MQCO_DELETE_PURGE are all treated as MQCO_NONE when specified on an MQCLOSE call for the corresponding MQOPEN call that created the queue.

Properties of permanent dynamic queues:

Permanent dynamic queues have the following properties:

- They hold persistent or nonpersistent messages.
- They are recoverable in the event of system failures.
- They are deleted when an application (not necessarily the one that issued the MQOPEN call that created the queue) successfully closes the queue using the MQCO_DELETE or MQCO_DELETE_PURGE option.
  - A close request with the MQCO_DELETE option fails if there are any messages (committed or uncommitted) still on the queue. A close request with the MQCO_DELETE_PURGE option succeeds even if there are committed messages on the queue (the messages being deleted as part of the close), but fails if there are any uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue.
  - If the delete request is successful, but the queue happens to be in use (by the creating, or another application), the queue is marked as being logically deleted and is only physically deleted when closed by the last application using the queue.
- They are not deleted if closed by an application that is not authorized to delete the queue, unless the closing application issued the MQOPEN call that created the queue. Authorization checks are performed against the user identifier (or alternate user identifier if MQOO_ALTERNATE_USER_AUTHORITY was specified) that was used to validate the corresponding MQOPEN call.
- They can be deleted in the same way as a normal queue.

Uses of dynamic queues:

You can use dynamic queues for:

- Applications that do not require queues to be retained after the application has terminated.
- Applications that require replies to messages to be processed by another application. Such applications can dynamically create a reply-to queue by opening a model queue. For example, a client application can:
  1. Create a dynamic queue.
  2. Supply its name in the *ReplyToQ* field of the message descriptor structure of the request message.
  3. Place the request on a queue being processed by a server.

The server can then place the reply message on the reply-to queue. Finally, the client could process the reply, and close the reply-to queue with the delete option.

Recommendations for uses of dynamic queues:

Consider the following points when using dynamic queues:

- In a client-server model, each client must create and use its own dynamic reply-to queue. If a dynamic reply-to queue is shared between more than one client, deleting the reply-to queue might be delayed because there is uncommitted activity outstanding against the queue, or because the queue is in use by another client. Additionally, the queue might be marked as being logically deleted, and inaccessible for subsequent API requests (other than MQCLOSE).
- If your application environment requires that dynamic queues must be shared between applications, ensure that the queue is only closed (with the delete option) when all activity against the queue has been committed. This should be by the last user. This ensures that deletion of the queue is not delayed, and minimizes the period that the queue is inaccessible because it has been marked as being logically deleted.

Transmission queues

When an application sends a message to a remote queue, the local queue manager stores the message in a special local queue, called a *transmission queue*.

A *message channel agent* (channel program), or *intra-group queuing agent* when using intra-group queuing on WebSphere MQ for z/OS, is associated with the transmission queue and the remote queue manager, and this delivers the message. When the message has been delivered, it is deleted from the transmission queue.

The message might have to pass through many queue managers (or *nodes*) on its journey to its final destination. There must be a transmission queue defined at each queue manager along the route, each holding messages waiting to be transmitted to the next node. (A shared transmission queue is used when using intra-group queuing on WebSphere MQ for z/OS.) There can be several transmission queues defined at a particular queue manager. A given transmission queue holds messages whose *next* destination is the same queue manager, although the messages might have different eventual destinations. There might also be several transmission queues for the same remote queue manager, with each one being used for a different type of service, for example.

Transmission queues can be used to trigger a message channel agent to send messages onward. For information about this, see "Starting WebSphere MQ applications using triggers" on page 195. These attributes are defined in the transmission queue definition (for triggered channels) or the process definition object (see "Process definitions" on page 59).

Initiation queues

An *initiation queue* is a local queue on which the queue manager puts a trigger message when a trigger event occurs on an application queue.

A trigger event is an event (for example, more than 10 messages arriving) that an application designer intends the queue manager to use as a cue, or trigger, to start a program to process the queue. For more information on how triggering works, see "Starting WebSphere MQ applications using triggers" on page 195.

Dead-letter (undelivered message) queues

A *dead-letter (undelivered message) queue* is a local queue on which the queue manager puts messages that it cannot deliver.

When the queue manager puts a message on the dead-letter queue, it adds a header to the message. This includes such information as the intended destination

of the original message, the reason that the queue manager put the message on the dead-letter queue, and the date and time that it did this.

Applications can also use the queue for messages that they cannot deliver. For more information, see "Using the dead-letter (undelivered message) queue" on page 66.

System command queues

These queues receive the PCF, MQSC, and CL commands, as supported on your platform, in readiness for the queue manager to action them.

On WebSphere MQ for z/OS the queue is known as the **SYSTEM.COMMAND.INPUT.QUEUE**; on other platforms it is known as the **SYSTEM.ADMIN.COMMAND.QUEUE**. The commands accepted vary by platform. See *WebSphere MQ Programmable Command Formats and Administration Interface* for details.

System default queues

The *system default queues* contain the initial definitions of the queues for your system. When you create a new queue, the queue manager copies the definition from the appropriate system default queue.

Administrative topic objects

An *administrative topic object* is a WebSphere MQ object that allows you to assign specific, non-default attributes to topics.

A *topic* is defined by an application publishing or subscribing to a particular *topic string*. A topic string can specify a hierarchy of topics by separating them with a forward slash character (/). This can be visualized by a *topic tree*. For example, if an application publishes to the topic strings `/Sport/American Football` and `/Sport/Soccer`, a topic tree will be created that has a parent node `Sport` with two children, `American Football`, and `Soccer`.

Topics inherit their attributes from the first parent administrative node found in their topic tree. If there are no administrative topic nodes in a particular topic tree, then all topics will inherit their attributes from the base topic object, SYSTEM.BASE.TOPIC.

You can create an administrative topic object at any node in a topic tree by specifying that node's topic string in the TOPICSTR attribute of the administrative topic object. You can also define other attributes for the administrative topic node. For more information about these attributes, see the *WebSphere MQ Script (MQSC) Command Reference*, or the *WebSphere MQ Programmable Command Formats and Administration Interface*. Each administrative topic object will, by default, inherit its attributes from its closest parent administrative topic node.

Administrative topic objects can also be used to hide the full topic tree from application developers. If an administrative topic object named `FOOTBALL.US` is created for the topic `/Sport/American Football`, an application can publish or subscribe to the object name `FOOTBALL.US` instead of the string `/Sport/American Football` with the same result.

If you enter a #, +, /, or * character within a topic string on a topic object, the character is treated as a normal character within the string, and is considered to be part of the topic string associated with an administrative topic object.

For more information about administrative topic objects, see the *WebSphere MQ Publish/Subscribe User's Guide*.

Namelists

A *namelist* is a WebSphere MQ object that contains a list of cluster names, queue names or authentication information object names. In a cluster, it can be used to identify a list of clusters for which the queue manager holds the repositories.

You can define and modify namelists only using the operations and control panels of WebSphere MQ for z/OS or MQSC commands.

Programs can use the MQI to find out which queues are included in these namelists. The organization of the namelists is the responsibility of the application designer and system administrator.

For a full description of the attributes of namelists, see the *WebSphere MQ Application Programming Reference*.

Process definitions

To allow an application to be started without the need for operator intervention (described in "Starting WebSphere MQ applications using triggers" on page 195), the attributes of the application must be known to the queue manager. These attributes are defined in a *process definition object*.

The `ProcessName` attribute is fixed when the object is created; you can change the others using the WebSphere MQ commands or the WebSphere MQ for z/OS operations and control panels. You can inquire about the values of *all* the attributes using the MQINQ call.

For a full description of the attributes of process definitions, see the *WebSphere MQ Application Programming Reference*.

Authentication information objects

An authentication information object contains authentication information used in Secure Sockets Layer (SSL) encrypted transport of information.

An authentication information object of AUTHTYPE CRLLDAP provides the definitions required to perform Certificate Revocation List (CRL) checking using LDAP servers. CRLs allow Certification Authorities to revoke certificates that can no longer be trusted.

For a full description of the attributes of authentication information objects, see the *WebSphere MQ Application Programming Reference*. For more information about SSL, see *WebSphere MQ Security*.

Channels

A *channel* is a communication link used by distributed queue managers.

There are two categories of channel in WebSphere MQ:

- *Message* channels, which are unidirectional, and transfer messages from one queue manager to another.

- *MQI* channels, which are bidirectional, and transfer MQI calls from a WebSphere MQ client to a queue manager, and responses from a queue manager to a WebSphere MQ client.

You need to consider these when designing your application, but programs are unaware of WebSphere MQ channel objects. For more information, see *WebSphere MQ Intercommunications* and *WebSphere MQ Clients*.

Storage classes

Supported only on WebSphere MQ for z/OS.

A *storage class* maps one or more queues to a page set. This means that messages for that queue are stored (subject to buffering) on that page set.

For further information about storage classes, see the WebSphere MQ for z/OS Concepts and Planning Guide.

Listeners

Listeners are processes that accept network requests from other queue managers, or client applications, and start associated channels.

Listener processes can be started using the **runmqlsr** control command. Listeners are available on all platforms.

Listener objects are WebSphere MQ objects that allow you to manage the starting and stopping of listener processes from within the scope of a queue manager. **Listener objects are not supported on WebSphere MQ for z/OS.** By defining attributes of a listener object you do the following:

- Configure the listener process.
- Specify whether the listener process automatically starts and stops when the queue manager starts and stops.

Services

Service objects are a way of defining programs to be executed when a queue manager starts or stops.

Not supported on WebSphere MQ for z/OS.

The programs can be split into the following types:

Servers

A *server* is a service object that has the parameter SERVTYPE specified as SERVER. A server service object is the definition of a program that will be executed when a specified queue manager is started. Only one instance of a server process can be executed concurrently. While running, the status of a server process can be monitored using the MQSC command, DISPLAY SVSTATUS. Typically server service objects are definitions of programs such as dead letter handlers or trigger monitors, however the programs that can be run are not limited to those supplied with WebSphere MQ. Additionally, a server service object can be defined to include a command that will be run when the specified queue manager is shutdown to end the program.

Commands

A *command* is a service object that has the parameter SERVTYPE specified as COMMAND. A command service object is the definition of a program that will be executed when a specified queue manager is started or stopped. Multiple instances of a command process can be executed concurrently. Command service objects differ from server service objects in that once the program is executed the queue manager will not monitor the program. Typically command service objects are definitions of programs that are short lived and will perform a specific task such as starting one, or more, other tasks.

Rules for naming WebSphere MQ objects

A WebSphere MQ queue, process definition, namelist, and channel can all have the same name. However, a WebSphere MQ object cannot have the same name as any other object of the same type. Names in WebSphere MQ are case sensitive.

The character set to use for naming all WebSphere MQ objects is as follows:

- Uppercase A–Z
- Lowercase a–z (but there are restrictions on the use of lowercase letters for z/OS console support)

  On systems using EBCDIC Katakana you cannot use lowercase characters.
- Numerics 0–9
- Period (.)
- Forward slash (/)
- Underscore (_)
- Percent sign (%)

Note:

1. Leading or embedded blanks are not allowed.
2. Avoid using names with leading or trailing underscores, because they cannot be handled by the WebSphere MQ for z/OS operations and control panels.
3. Any name that is less than the full field length can be padded to the right with blanks. All short names that are returned by the queue manager are always padded to the right with blanks.
4. Any structure to the names (for example, the use of the period or underscore) is not significant to the queue manager.
5. On i5/OS systems, within CL, lowercase a-z, forward slash (/), and percent (%) are special characters. If you use any of these characters in a name, enclose the name in quotation marks. Lowercase a-z characters are changed to uppercase if the name is not enclosed in quotation marks.
6. On Windows systems, the first character of a queue manager name cannot be a forward slash (/).

Queue names

The name of a queue has two parts:

- The name of a queue manager
- The local name of the queue as it is known to that queue manager

Each part of the queue name is 48 characters long.

To refer to a local queue, you can omit the name of the queue manager (by replacing it with blank characters or using a leading null character). However, all queue names returned to a program by WebSphere MQ contain the name of the queue manager.

A shared queue, accessible to any queue manager in its queue-sharing group, cannot have the same name as any non-shared local queue in the same queue-sharing group. This restriction avoids the possibility of an application mistakenly opening a shared queue when it intended to open a local queue, or *vice versa.* Shared queues and queue-sharing groups are only available on WebSphere MQ for z/OS.

To refer to a remote queue, a program must include the name of the queue manager in the full queue name, or there must be a local definition of the remote queue.

When an application uses a queue name, that name can be either the name of a local queue (or an alias to one) or the name of a local definition of a remote queue, but the application does not need to know which, unless it needs to get a message from the queue (when the queue must be local). When the application opens the queue object, the MQOPEN call performs a name resolution function to determine on which queue to perform subsequent operations. The significance of this is that the application has no built-in dependency on particular queues being defined at particular locations in a network of queue managers. Therefore, if a system administrator relocates queues in the network, and changes their definitions, the applications that use those queues do not need to be changed.

Process definition, authentication information object, and namelist names

Process definitions, authentication information objects, and namelists can have names up to 48 characters long.

Channel names

Channels can have names up to 20 characters long.

See *WebSphere MQ Intercommunications* for further information on channels.

Reserved object names

Names that start with SYSTEM. are reserved for objects defined by the queue manager.

Handling program errors

Your application can encounter errors associated with its MQI calls either when it makes a call or when its message is delivered to its final destination:

- Whenever possible, the queue manager returns any errors as soon as an MQI call is made. These are *locally determined errors*.
- When sending messages to a remote queue, errors might not be apparent when the MQI call is made. In this case, the queue manager that identifies the errors reports them by sending another message to the originating program. These are *remotely determined errors*.

This chapter gives advice on how to handle both types of error, under these headings:

- "Locally determined errors" on page 63

- "Using report messages for problem determination" on page 65
- "Remotely determined errors" on page 66

Locally determined errors

The three most common causes of errors that the queue manager can report immediately are:

- Failure of an MQI call; for example, because a queue is full
- An interruption to the running of some part of the system on which your application depends; for example, the queue manager
- Messages containing data that cannot be processed successfully

If you are using the asynchronous put facility, errors are not reported immediately. Use the MQSTAT call to retrieve status information about previous asynchronous put operations.

Failure of an MQI call

The queue manager can report immediately any errors in the coding of an MQI call. It does this using a set of predefined return codes. These are divided into completion codes and reason codes.

To show whether or not a call is successful, the queue manager returns a *completion code* when the call completes. There are three completion codes, indicating success, partial completion, and failure of the call. The queue manager also returns a *reason code* that indicates the reason for the partial completion or the failure of the call.

The completion and reason codes for each call are listed with the description of that call in the *WebSphere MQ Application Programming Reference*. For more detailed information, including ideas for corrective action, see:

- *WebSphere MQ for z/OS Messages and Codes* for WebSphere MQ for z/OS
- *WebSphere MQ Messages* for all other WebSphere MQ platforms

Design your programs to handle all the return codes that can arise from each call.

System interruptions

Your application might be unaware of any interruption if the queue manager to which it is connected has to recover from a system failure. However, you must design your application to ensure that your data is not lost if such an interruption occurs.

The methods that you can use to make sure that your data remains consistent depends on the platform on which your queue manager is running:

z/OS In the CICS and IMS environments, you can make MQPUT and MQGET calls within units of work that are managed by CICS or IMS. In the batch environment, you can make MQPUT and MQGET calls in the same way, but you must declare syncpoints using:

- The WebSphere MQ for z/OS MQCMIT and MQBACK calls (see "Committing and backing out units of work" on page 183), or
- The z/OS Transaction Management and Recoverable Resource Manager Services (RRS) to provide two-phase syncpoint support. RRS allows you to update both WebSphere MQ and other RRS-enabled product resources, such as DB2 stored procedure resources, within a single

logical unit of work. For information on RRS syncpoint support see "Transaction management and recoverable resource manager services" on page 187.

i5/OS You can make your MQPUT and MQGET calls within global units of work that are managed by i5/OS commitment control. You can declare syncpoints by using the native i5/OS COMMIT and ROLLBACK commands or the language-specific commands. Local units of work are managed by WebSphere MQ using the MQCMIT and MQBACK calls.

UNIX systems and Windows systems
In these environments, you can make your MQPUT and MQGET calls in the usual way, but you must declare syncpoints by using the MQCMIT and MQBACK calls (see "Committing and backing out units of work" on page 183). In the CICS environment, MQCMIT and MQBACK commands are disabled, because you can make your MQPUT and MQGET calls within units of work that are managed by CICS.

Use persistent messages for carrying all data that you cannot afford to lose. Persistent messages are reinstated on queues if the queue manager has to recover from a failure. With WebSphere MQ on UNIX systems and WebSphere MQ for Windows, an MQGET or MQPUT call within your application will fail at the point of filling all the log files, with the message MQRC_RESOURCE_PROBLEM. For more information on log files on AIX, HP-UX, Linux, Solaris, and Windows systems, see the *WebSphere MQ System Administration Guide*; for z/OS see the *WebSphere MQ for z/OS Concepts and Planning Guide*.

If the queue manager is stopped by an operator while an application is running, the quiesce option is usually used. The queue manager enters a quiescing state in which applications can continue to do work, but they must terminate as soon as convenient. Small, quick applications can probably ignore the quiescing state and continue until they terminate as normal. Longer running applications, or ones that wait for messages to arrive, should use the *fail if quiescing* option when they use the MQOPEN, MQPUT, MQPUT1, and MQGET calls. These options mean that the calls fail when the queue manager quiesces, but the application might still have time to terminate cleanly by issuing calls that ignore the quiescing state. Such applications could also commit, or back out, changes that they have made, and then terminate.

If the queue manager is forced to stop (that is, stop without quiescing), applications will receive the MQRC_CONNECTION_BROKEN reason code when they make MQI calls. At this point, exit the application or, alternatively, on WebSphere MQ for i5/OS, WebSphere MQ on UNIX systems, and WebSphere MQ for Windows, issue an MQDISC call.

Messages containing incorrect data

When you use units of work in your application, if a program cannot successfully process a message that it retrieves from a queue, the MQGET call is backed out.

The queue manager maintains a count (in the *BackoutCount* field of the message descriptor) of the number of times that happens. It maintains this count in the descriptor of each message that is affected. This count can provide valuable information about the efficiency of an application. Messages whose backout counts are increasing over time are being repeatedly rejected; design your application so that it analyzes the reasons for this and handles such messages accordingly.

On WebSphere MQ for z/OS, to make the backout count survive restarts of the queue manager, set the *HardenGetBackout* attribute to MQQA_BACKOUT_HARDENED; otherwise, if the queue manager has to restart, it does not maintain an accurate backout count for each message. Setting the attribute this way adds the penalty of extra processing.

On WebSphere MQ for i5/OS, WebSphere MQ for Windows, and WebSphere MQ on UNIX systems, the backout count always survives restarts of the queue manager.

Also, on WebSphere MQ for z/OS, when you remove messages from a queue within a unit of work, you can mark one message so that it is *not* made available again if the unit of work is backed out *by the application*. The marked message is treated as if it has been retrieved under a new unit of work. You mark the message that is to skip backout using the MQGMO_MARK_SKIP_BACKOUT option (in the MQGMO structure) when you use the MQGET call. See "Skipping backout" on page 152 for more information about this technique.

Using report messages for problem determination

The remote queue manager cannot report errors such as failing to put a message on a queue when you make your MQI call, but it can send you a report message to say how it has processed your message.

Within your application you can create (MQPUT) report messages as well as select the option to receive them (in which case they are sent by either another application or by a queue manager).

Creating report messages

Report messages enable an application to tell another application that it cannot deal with the message that was sent.

However, the *Report* field must initially be analyzed to determine whether the application that sent the message is interested in being informed of any problems. Having determined that a report message is required, you have to decide:

- Whether you want to include the entire original message, just the first 100 bytes of data, or none of the original message.
- What to do with the original message. You can discard it or let it go to the dead-letter queue.
- Whether the contents of the *MsgId* and *CorrelId* fields are needed as well.

Use the *Feedback* field to indicate the reason for the report message being generated. Put your report messages on an application's reply-to queue. Refer to the *WebSphere MQ Application Programming Reference* for further information.

Requesting and receiving (MQGET) report messages:

When you send a message to another application, you are not informed of any problems unless you complete the *Report* field to indicate the feedback that you require. The options available to you are in the *WebSphere MQ Application Programming Reference*.

Queue managers always put report messages on an application's reply-to queue and it is recommended that your own applications do the same. When you use the report message facility, specify the name of your reply-to queue in the message descriptor of your message; otherwise, the MQPUT call fails.

Your application must contain procedures that monitor your reply-to queue and process any messages that arrive on it. Remember that a report message can contain all the original message, the first 100 bytes of the original message, or none of the original message.

The queue manager sets the *Feedback* field of the report message to indicate the reason for the error; for example, the target queue does not exist. Your programs should do the same.

For more information on report messages, see "Report messages" on page 19.

Remotely determined errors

When you send messages to a remote queue, even when the local queue manager has processed your MQI call without finding an error, other factors can influence how your message is handled by a remote queue manager.

For example, the queue that you are targeting might be full, or might not even exist. If your message has to be handled by other intermediate queue managers on the route to the target queue, any of these could find an error.

Problems delivering a message

When an MQPUT call fails, you can try to put the message on the queue again, return it to the sender, or put it on the dead-letter queue.

Each option has its merits, but you might not want to retry putting a message if the reason that the MQPUT failed was because the destination queue was full. In this instance, putting it on the dead-letter queue allows you to deliver it to the correct destination queue later on.

Retry message delivery:

Before the message is put on a dead-letter queue, a remote queue manager attempts to put the message on the queue again if the attributes *MsgRetryCount* and *MsgRetryInterval* have been set for the channel, or if there is a retry exit program for it to use (the name of which is held in the channel attribute *MsgRetryExitId* field).

If the *MsgRetryExitId* field is blank, the values in the attributes *MsgRetryCount* and *MsgRetryInterval* are used.

If the *MsgRetryExitId* field is not blank, the exit program of this name runs. For more information on using your own exit programs, see *WebSphere MQ Intercommunications*.

Return message to sender:

You return a message to the sender by requesting a report message to be generated to include all of the original message.

See "Report messages" on page 19 for details on report message options.

Using the dead-letter (undelivered message) queue

When a queue manager cannot deliver a message, it attempts to put the message on its dead-letter queue. This queue should be defined when the queue manager is installed.

Your programs can use the dead-letter queue in the same way that the queue manager uses it. You can find the name of the dead-letter queue by opening the queue manager object (using the MQOPEN call) and inquiring about the *DeadLetterQName* attribute (using the MQINQ call).

When the queue manager puts a message on this queue, it adds a header to the message, the format of which is described by the dead-letter header (MQDLH) structure, in the *WebSphere MQ Application Programming Reference*. This header includes the name of the target queue and the reason that the message was put on the dead-letter queue. It must be removed and the problem must be resolved before the message is put on the intended queue. Also, the queue manager changes the *Format* field of the message descriptor (MQMD) to indicate that the message contains an MQDLH structure.

MQDLH structure

You are recommended to add an MQDLH structure to all messages that you put on the dead-letter queue; however, if you intend to use the dead-letter handler provided by certain WebSphere MQ products, you ***must*** add an MQDLH structure to your messages.

The addition of the header to a message might make the message too long for the dead-letter queue, so always make sure that your messages are shorter than the maximum size allowed for the dead-letter queue, by at least the value of the MQ_MSG_HEADER_LENGTH constant. The maximum size of messages allowed on a queue is determined by the value of the *MaxMsgLength* attribute of the queue. For the dead-letter queue, make sure that this attribute is set to the maximum allowed by the queue manager. If your application cannot deliver a message, and the message is too long to be put on the dead-letter queue, follow the advice given in the description of the MQDLH structure.

Ensure that the dead-letter queue is monitored, and that any messages arriving on it get processed. The dead-letter queue handler runs as a batch utility and can be used to perform various actions on selected messages on the dead-letter queue. For further details, see *WebSphere MQ System Administration Guide* for WebSphere MQ for AIX, HP-UX, Linux, Solaris, and Windows systems; for WebSphere MQ for z/OS see *WebSphere MQ for z/OS System Administration Guide*; for i5/OS see *WebSphere MQ for i5/OS System Administration Guide*.

If data conversion is necessary, the queue manager converts the header information when you use the MQGMO_CONVERT option on the MQGET call. If the process putting the message is an MCA, the header is followed by all the text of the original message.

Messages put on the dead-letter queue might be truncated if they are too long for this queue. A possible indication of this situation is the messages on the dead-letter queue being the same length as the value of the *MaxMsgLength* attribute of the queue.

Dead-letter queue processing:

The rest of this chapter contains general-use programming interface information.

Dead-letter queue processing depends on local system requirements, but consider the following when you draw up the specification:

- The message can be identified as having a dead-letter queue header because the value of the format field in the MQMD, is MQFMT_DEAD_LETTER_HEADER.
- On WebSphere MQ for z/OS using CICS, if an MCA puts this message to the dead-letter queue, the *PutApplType* field is MQAT_CICS, and the *PutApplName* field is the *ApplId* of the CICS system followed by the transaction name of the MCA.
- The reason for the message to be routed to the dead-letter queue is contained in the *Reason* field of the dead-letter queue header.
- The dead-letter queue header contains details of the destination queue name and queue manager name.
- The dead-letter queue header contains fields that have to be reinstated in the message descriptor before the message is put to the destination queue. These are:
  1. *Encoding*
  2. *CodedCharSetId*
  3. *Format*
- The message descriptor is the same as PUT by the original application, except for the three fields shown above.

Your dead-letter queue application must do one or more of the following:

- Examine the *Reason* field. A message might have been put by an MCA for the following reasons:
  - The message was longer than the maximum message size for the channel

    The reason is MQRC_MSG_TOO_BIG_FOR_CHANNEL (or MQRC_MSG_TOO_BIG_FOR_Q_MGR if you are using CICS for distributed queuing on WebSphere MQ for z/OS)
  - The message could not be put to its destination queue

    The reason is any MQRC_* reason code that can be returned by an **MQPUT** operation
  - A user exit has requested this action

    The reason code is that supplied by the user exit, or the default MQRC_SUPPRESSED_BY_EXIT
- Try to forward the message to its intended destination, where this is possible.
- Retain the message for a certain length of time before discarding when the reason for the diversion is determined, but not immediately correctable.
- Give instructions to administrators correct problems where these have been determined.
- Discard messages that are corrupted or otherwise not processible.

There are two ways to deal with the messages that you have recovered from the dead-letter queue:

1. If the message is for a local queue:
   - Carry out any code translations required to extract the application data
   - Carry out code conversions on that data if this is a local function
   - Put the resulting message on the local queue with all the detail of the message descriptor restored
2. If the message is for a remote queue, put the message on the queue.

For information on how undelivered messages are handled in a distributed queuing environment, see *WebSphere MQ Intercommunications*.

Chapter 2. Writing a WebSphere MQ application

Introducing the Message Queue Interface

This chapter introduces the features of the Message Queue Interface (MQI).

The remaining chapters in this part of the book describe how to use these features. Detailed descriptions of the calls, structures, data types, return codes, and constants are given in the *WebSphere MQ Application Programming Reference*.

The MQI is introduced under these headings:

What is in the MQI?

The Message Queue Interface consists of the following:

- *Calls* through which programs can access the queue manager and its facilities
- *Structures* that programs use to pass data to, and get data from, the queue manager
- *Elementary data types* for passing data to, and getting data from, the queue manager

WebSphere MQ for z/OS also supplies:

- Two extra calls through which z/OS batch programs can commit and back out changes.
- *Data definition files* (sometimes known as copy files, macros, include files, and header files) that define the values of constants supplied with WebSphere MQ for z/OS.
- *Stub programs* to link-edit to your applications.
- A suite of sample programs that demonstrate how to use the MQI on the z/OS platform. For further information about these samples, see "Sample programs for WebSphere MQ for z/OS" on page 452.

WebSphere MQ for i5/OS also supplies:

- *Data definition files* (sometimes known as copy files, macros, include files, and header files) that define the values of constants supplied with WebSphere MQ for i5/OS.
- Three stub programs to link-edit to your ILE C, ILE COBOL, and ILE RPG applications.
- A suite of sample programs that demonstrate how to use the MQI on the i5/OS platform. For further information about these samples, see "Sample programs (all platforms except z/OS)" on page 395.

WebSphere MQ for Windows and WebSphere MQ on UNIX systems also supply:

- Calls through which WebSphere MQ for Windows and WebSphere MQ on UNIX systems programs can commit and back out changes.
- *Include files* that define the values of constants supplied on these platforms.
- *Library files* to link your applications.
- A suite of sample programs that demonstrate how to use the MQI on these platforms.
- Sample source and executable code for bindings to external transaction managers.

Calls

The calls in the MQI can be grouped as follows:

MQCONN, MQCONNX, and MQDISC
Use these calls to connect a program to (with or without options), and disconnect a program from, a queue manager. If you write CICS programs for z/OS, you do not need to use these calls. However, you are recommended to use them if you want to port your application to other platforms.

MQOPEN and MQCLOSE
Use these calls to open and close an object, such as a queue.

MQPUT and MQPUT1
Use these calls to put a message on a queue.

MQGET
Use this call to browse messages on a queue, or to remove messages from a queue.

MQSUB, MQSUBRQ
Use these calls to register a subscription to a topic, and to request publications matching the subscription.

MQINQ
Use this call to inquire about the attributes of an object.

MQSET
Use this call to set some of the attributes of a queue. You cannot set the attributes of other types of object.

MQBEGIN, MQCMIT, and MQBACK
Use these calls when WebSphere MQ is the coordinator of a unit of work. MQBEGIN starts the unit of work. MQCMIT and MQBACK end the unit of work, either committing or rolling back the updates made during the unit of work. i5/OS commitment controller is used to coordinate global units of work on i5/OS. Native start commitment control, commit, and rollback commands are used.

MQCRTMH, MQBUFMH, MQMHBUF, MQDLTMH
Use these calls to create a message handle, to convert a message handle to a buffer or a buffer to a message handle, and to delete a message handle.

MQSETMP, MQINQMP, MQDLTMP
Use these calls to set a message property on a message handle, inquire on a message property, and delete a property from a message handle.

MQCB, MQCB_FUNCTION, MQCTL
Use these calls to register and control a callback function.

MQSTAT
Use this call to retrieve status information about previous asynchronous put operations.

The MQI calls are described fully in the *WebSphere MQ Application Programming Reference*

Syncpoint calls

Syncpoint calls are available as follows:

WebSphere MQ for z/OS calls:

WebSphere MQ for z/OS provides the MQCMIT and MQBACK calls.

Use these calls in z/OS batch programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in other environments:

CICS Use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

IMS Use the IMS syncpoint facilities, such as the GU (get unique) to the IOPCB, CHKP (checkpoint), and ROLB (rollback) calls.

RRS Use MQCMIT and MQBACK or SRRCMIT and SRRBACK as appropriate. (See "Transaction management and recoverable resource manager services" on page 187.)

Note: SRRCMIT and SRRBACK are native RRS commands, they are not MQI calls.

For backward compatibility, the CSQBCMT and CSQBBAK calls are available as synonyms for MQCMIT and MQBACK. These are described in the *WebSphere MQ Application Programming Reference*.

i5/OS calls:

WebSphere MQ for i5/OS provides the MQCMIT and MQBACK commands. You can also use the i5/OS COMMIT and ROLLBACK commands, or any other commands or calls that initiate the i5/OS commitment control facilities (for example, EXEC CICS SYNCPOINT).

WebSphere MQ calls on other platforms:

The following products provide the MQCMIT and MQBACK calls:

- WebSphere MQ for Windows
- WebSphere MQ on UNIX systems

Use syncpoint calls in programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in the CICS environment, use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

Data conversion

The MQXCNVC (convert characters) call converts message character data from one character set to another. Except on WebSphere MQ for z/OS, this call is used only from a data-conversion exit.

See the *WebSphere MQ Application Programming Reference* for the syntax used with the MQXCNVC call, and "Writing data-conversion exits" on page 163 for guidance on writing and invoking data conversion exits.

Structures

Structures, used with the MQI calls listed in "Calls" on page 70, are supplied in data definition files for each of the supported programming languages. WebSphere MQ for z/OS and WebSphere MQ for i5/OS supply files that contain constants for you to use when filling in some of the fields of these structures. For more information on these, see "WebSphere MQ data definitions."

All the structures are described in the *WebSphere MQ Application Programming Reference*.

Elementary data types

For the supported programming languages, the MQI provides elementary data types or unstructured fields.

These data types are described fully in the *WebSphere MQ Application Programming Reference*.

WebSphere MQ data definitions

WebSphere MQ for z/OS supplies data definitions in the form of COBOL copy files, assembler-language macros, a single PL/I include file, a single C language include file, and C++ language include files.

WebSphere MQ for i5/OS supplies data definitions in the form of COBOL copy files, RPG copy files, C language include files, and C++ language include files.

The data definition files supplied with WebSphere MQ contain:

- Definitions of all the WebSphere MQ constants and return codes
- Definitions of the WebSphere MQ structures and data types
- Constant definitions for initializing the structures
- Function prototypes for each of the calls (for PL/I and the C language only)

For a full description of WebSphere MQ data definition files, see Chapter 9, "WebSphere MQ data definition files," on page 575.

WebSphere MQ stub programs and library files

The stub programs and library files provided are listed here, for each platform.

For more information about how to use stub programs and library files when you build an executable application, see Chapter 3, "Building a WebSphere MQ application," on page 339. For information about linking to C++ library files, see *WebSphere MQ Using C++*.

WebSphere MQ for z/OS:

Before you can run a program written with WebSphere MQ for z/OS, you must link-edit it to the stub program supplied with WebSphere MQ for z/OS for the environment in which you are running the application.

The stub program provides the first stage of the processing of your calls into requests that WebSphere MQ for z/OS can process.

WebSphere MQ for z/OS supplies the following stub programs:

CSQBSTUB	Stub program for z/OS batch programs
CSQBRRSI	Stub program for z/OS batch programs using RRS via the MQI
CSQBRSTB	Stub program for z/OS batch programs using RRS directly
CSQCSTUB	Stub program for CICS programs
CSQQSTUB	Stub program for IMS programs
CSQXSTUB	Stub program for distributed queuing non-CICS exits
CSQASTUB	Stub program for data-conversion exits

Note: If you use the CSQBRSTB stub program, link-edit with ATRSCSS from SYS1.CSSLIB. (SYS1.CSSLIB is also known as the *Callable Services Library*). For more information about RRS see "Transaction management and recoverable resource manager services" on page 187.

Alternatively, you can dynamically call the stub from within your program. This technique is described in "Dynamically calling the WebSphere MQ stub" on page 377.

In IMS, you might also need to use a special language interface module that is supplied by WebSphere MQ.

WebSphere MQ for i5/OS:

In WebSphere MQ for i5/OS, link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

For non-threaded applications:

AMQZSTUB	Server service program provided for compatibility with releases before V5R1M0
AMQVSTUB	Data conversion service program provided for compatibility with releases before V5R1M0
LIBMQM	Server service program
LIBMQIC	Client service program
IMQB23I4	C++ base service program
IMQS23I4	C++ server service program
LIBMQMZF	Installable exits for C

In a threaded application:

LIBMQM_R	Server service program
IMQB23I4_R	C++ base service program
IMQS23I4_R	C++ server service program
LIBMQMZF_R	Installable exits for C

On WebSphere MQ for i5/OS you can write your applications in C++. To see how to link your C++ applications, and for full details of all aspects of using C++, see *WebSphere MQ Using C++*.

WebSphere MQ for Windows:

On WebSphere MQ for Windows, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

Library file and location	Purpose
install_location\Tools\Lib\ mqm.lib	Server for C (32-bit)
install_location\Tools\Lib\ mqic.lib	Client for C (32-bit)
install_location\Tools\Lib\ mqmxa.lib	Server XA interface for C (32-bit)
install_location\Tools\Lib\ mqcxa.lib	Client XA interface for C (32-bit)
install_location\Tools\Lib\ mqicxa.lib	Client MTS for C (32-bit)
install_location\Tools\Lib\ mqmcics4.lib	Server TXSeries CICS support for C (32-bit)
install_location\Tools\Lib\ mqccics4.lib	Client TXSeries CICS support for C (32-bit)
install_location\Tools\Lib\ mqmzf.lib	Installable services exits for C (32-bit)
install_location\Tools\Lib\ mqmcbb.lib	Server for IBM COBOL (32-bit)
install_location\Tools\Lib\ mqmcb.lib	Server for Micro Focus COBOL (32-bit)
install_location\Tools\Lib\ mqiccbb.lib	Client for IBM COBOL (32-bit)
install_location\Tools\Lib\ mqiccb.lib	Client for Micro Focus COBOL (32-bit)
install_location\Tools\Lib\ imqs23vn.lib	Server for C++ (32-bit)
install_location\Tools\Lib\ imqc23vn.lib	Client for C++ (32-bit)
install_location\Tools\Lib\ imqb23vn.lib	Base for C++ (32-bit)
install_location\Tools\Lib\ imqx23vn.lib	Client MTS for C++ (32-bit)
install_location\Tools\Lib64\ mqm.lib	Server for C (64-bit)
install_location\Tools\Lib64\ mqic.lib	Client for C (64-bit)
install_location\Tools\Lib64\ mqmxa.lib	Server XA interface for C (64-bit)
install_location\Tools\Lib64\ mqcxa.lib	Client XA interface for C (64-bit)
install_location\Tools\Lib64\ mqicxa.lib	Client MTS for C (64-bit)
install_location\Tools\Lib64\ mqmcbb.lib	Server for IBM COBOL (64-bit)
install_location\Tools\Lib64\ mqmcb.lib	Server for Micro Focus COBOL (64-bit)

install_location\Tools\Lib64\ mqiccbb.lib	Client for IBM COBOL (64-bit)
install_location\Tools\Lib64\ mqiccb.lib	Client for Micro Focus COBOL (64-bit)
install_location\Tools\Lib64\ imqs23vn.lib	Server for C++ (64-bit)
install_location\Tools\Lib64\ imqc23vn.lib	Client for C++ (64-bit)
install_location\Tools\Lib64\ imqb23vn.lib	Base for C++ (64-bit)
install_location\Tools\Lib64\ imqx23vn.lib	Client MTS for C++ (64-bit)

These files are shipped for compatibility with previous releases:

mqic32.lib

mqic32xa.lib

WebSphere MQ for AIX:

On WebSphere MQ for AIX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application:

libmqm.a	Server for C
libmqic.a	Client for C
libmqmzf.a	Installable service exits for C
libmqmxa.a	Server XA interface for C
libmqmxa64.a	Server alternative XA interface for C
libmqcxa.a	Client XA interface for C
libmqcxa64.a	Client alternative XA interface for C
libmqmcbrt.o	WebSphere MQ run-time library for Micro Focus COBOL support
libmqmcb.a	Server for COBOL
libmqicb.a	Client for COBOL
libimqc23ia.a	Client for C++
libimqs23ia.a	Server for C++

In a threaded application:

libmqm_r.a	Server for C
libmqic_r.a	Client for C
libmqmzf_r.a	Installable service exits for C
libmqmxa_r.a	Server XA interface for C
libmqmxa64_r.a	Server alternative XA interface for C
libmqcxa_r.a	Client XA interface for C
libmqcxa64_r.a	Client alternative XA interface for C
libimqc23ia_r.a	Client for C++
libimqs23ia_r.a	Server for C++

WebSphere MQ for HP-UX:

On WebSphere MQ for HP-UX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

PA-RISC platform:

In a non-threaded application:

libmqm.sl	Server for C
libmqic.sl	Client for C
libmqmzf.sl	Installable service exits for C
libmqmxa.sl	Server XA interface for C
libmqmxa64.sl	Server alternative XA interface for C
libmqcxa.sl	Client XA interface for C
libmqcxa64.sl	Client alternative XA interface for C
libimqi23ah.sl	C++
libmqmcbrt.o	WebSphere MQ run-time library for Micro Focus COBOL support
libmqmcb.sl	Server for COBOL
libmqicb.sl	Client for COBOL

In a threaded application:

libmqm_r.sl	Server for C
libmqmzf_r.sl	Installable service exits for C
libmqmxa_r.sl	Server XA interface for C
libmqmxa64_r.sl	Server alternative XA interface for C
libmqcxa_r.sl	Client XA interface for C
libmqcxa64_r.sl	Client alternative XA interface for C
libimqi23ah_r.sl	C++

IA64 (IPF) platform:

In a non-threaded application:

libmqm.so	Server for C
libmqic.so	Client for C
libmqmzf.so	Installable service exits for C
libmqmxa.so	Server XA interface for C
libmqmxa64.so	Server alternative XA interface for C
libmqcxa.so	Client XA interface for C
libmqcxa64.so	Client alternative XA interface for C
libimqi23ah.so	C++
libmqmcbrt.o	WebSphere MQ run-time library for Micro Focus COBOL support
libmqmcb.so	Server for COBOL
libmqicb.so	Client for COBOL

In a threaded application:

libmqm_r.so	Server for C
libmqmzf_r.so	Installable service exits for C
libmqmxa_r.so	Server XA interface for C
libmqmxa64_r.so	Server alternative XA interface for C
libmqcxa_r.so	Client XA interface for C
libmqcxa64_r.so	Client alternative XA interface for C

libimqi23ah_r.so C++

WebSphere MQ for Linux:

On WebSphere MQ for Linux, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application:

libmqm.so Server for C
libmqic.so Client for C
libmqmzf.so Installable service exits for C
libmqmxa.so Server XA interface for C
libmqmxa64.so Server alternative XA interface for C
libmqcxa.so Client XA interface for C
libmqcxa64.so Client alternative XA interface for C
libimqc23gl.so Client for C++
libimqs23gl.so Server for C++

In a threaded application:

libmqm_r.so Server for C
libmqic_r.so Client for C
libmqmzf_r.so Installable service exits for C
libmqmxa_r.so Server XA interface for C
libmqmxa64_r.so Server alternative XA interface for C
libmqcxa_r.so Client XA interface for C
libmqcxa64_r.so Client alternative XA interface for C
libimqc23gl_r.so Client for C++
libimqs23gl_r.so Server for C++

WebSphere MQ for Solaris:

On WebSphere MQ for Solaris, you must link your program to the MQI library files supplied for the environment in which you are running your application in addition to those provided by the operating system.

libmqm.so Server for C
libmqmzse.so For C
libmqic.so Client for C
libmqmcs.so Common services for C
libmqmzf.so Installable service exits for C
libmqmxa.so Server XA interface for C
libmqmxa64.so Server alternative XA interface for C
libmqcxa.so Client XA interface for C
libmqcxa64.so Client alternative XA interface for C
libimqc23as.a Client for C++
libimqs23as.a Server for C++

Parameters common to all the calls

There are two types of parameter common to all the calls: handles and return codes.

Using handles

All MQI calls use one or more *handles*. These identify the queue manager, queue or other object, message, or subscription, as appropriate to the call.

For a program to communicate with a queue manager, the program must have a unique identifier by which it knows that queue manager. This identifier is called a *connection handle*, sometimes referred to as an *Hconn*. For CICS programs, the connection handle is always zero. For all other platforms or styles of programs, the connection handle is returned by the MQCONN or MQCONNX call when the program connects to the queue manager. Programs pass the connection handle as an input parameter when they use the other calls.

For a program to work with a WebSphere MQ object, the program must have a unique identifier by which it knows that object. This identifier is called an *object handle*, sometimes referred to as an *Hobj*. The handle is returned by the MQOPEN call when the program opens the object to work with it. Programs pass the object handle as an input parameter when they use subsequent MQPUT, MQGET, MQINQ, MQSET, or MQCLOSE calls.

Similarly, the MQSUB call returns a *subscription handle* or *Hsub*, which is used to identify the subscription in subsequent MQGET, MQCB or MQSUBRQ calls, and certain calls processing message properties use a *message handle* or *Hmsg*.

Understanding return codes

A completion code and a reason code are returned as output parameters by each call. These are known collectively as *return codes*.

To show whether or not a call is successful, each call returns a *completion code* when the call is complete. The completion code is usually either MQCC_OK or MQCC_FAILED, showing success and failure, respectively. Some calls can return an intermediate state, MQCC_WARNING, indicating partial success.

Each call also returns a *reason code* that shows the reason for the failure, or partial success, of the call. There are many reason codes, covering such circumstances as a queue being full, get operations not being allowed for a queue, and a particular queue not being defined for the queue manager. Programs can use the reason code to decide how to proceed. For example, they can prompt users to change their input data, then make the call again, or they can return an error message to the user.

When the completion code is MQCC_OK, the reason code is always MQRC_NONE.

The completion and reason codes for each call are listed with the description of that call in *WebSphere MQ Application Programming Reference*. For more detailed information, including ideas for corrective action, see:

- *WebSphere MQ for z/OS Messages and Codes* for WebSphere MQ for z/OS
- *WebSphere MQ Messages* for all other WebSphere MQ platforms

Specifying buffers

The queue manager refers to buffers only if they are required. If you do not require a buffer on a call or the buffer is zero in length, you can use a null pointer to a buffer.

Always use datalength when specifying the size of the buffer that you require.

When you use a buffer to hold the output from a call (for example, to hold the message data for an MQGET call, or the values of attributes queried by the MQINQ call), the queue manager attempts to return a reason code if the buffer you specify is not valid or is in read-only storage. However, it might not always be able to return a reason code.

Programming language considerations

WebSphere MQ provides support for the following programming languages:

- C
- C++ (see *WebSphere MQ Using C++* for information about coding WebSphere MQ programs in C++)
- Visual Basic (Windows systems only)
- COBOL
- Assembler language (WebSphere MQ for z/OS only)
- RPG (WebSphere MQ for i5/OS only)
- PL/I (WebSphere MQ for z/OS only)

The call interface, and how you can code the calls in each of these languages, is described in the *WebSphere MQ Application Programming Reference*.

WebSphere MQ provides data definition files to help you to write your applications. For a full description, see Chapter 9, "WebSphere MQ data definition files," on page 575.

If you can choose which language to code your programs in, consider the maximum length of the messages that your programs will process. If your programs will process only messages of a known maximum length, you can code them in any of the supported programming languages. But if you do not know the maximum length of the messages that the programs will have to process, the language you choose will depend on whether you are writing a CICS, IMS, or batch application:

IMS and batch
: Code the programs in C, PL/I, or assembler language to use the facilities these languages offer for obtaining and releasing arbitrary amounts of memory. Alternatively, you could code your programs in COBOL, but use assembler language, PL/I, or C subroutines to get and release storage.

CICS
: Code the programs in any language supported by CICS. The EXEC CICS interface provides the calls for managing memory, if necessary.

Coding in C

Note the information in the following sections when coding WebSphere MQ programs in C.

Parameters of the MQI calls:

Parameters that are *input-only* and of type MQHCONN, MQHOBJ, MQHMSG, or MQLONG are passed by value; for all other parameters, the *address* of the parameter is passed by value.

Not all parameters that are passed by address need to be specified every time a function is invoked. Where a particular parameter is not required, a null pointer

can be specified as the parameter on the function invocation, in place of the address of the parameter data. Parameters for which this is possible are identified in the call descriptions.

No parameter is returned as the value of the function; in C terminology, this means that all functions return void.

The attributes of the function are defined by the MQENTRY macro variable; the value of this macro variable depends on the environment.

Parameters with undefined data type:

The MQGET, MQPUT, and MQPUT1 functions each have a *Buffer* parameter that has an undefined data type. This parameter is used to send and receive the application's message data.

Parameters of this sort are shown in the C examples as arrays of MQBYTE. You can declare the parameters in this way, but it is usually more convenient to declare them as the structure that describes the layout of the data in the message. The function parameter is declared as a pointer-to-void, and so the address of any sort of data can be specified as the parameter on the function invocation.

Data types:

All data types are defined by means of the `typedef` statement.

For each data type, the corresponding pointer data type is also defined. The name of the pointer data type is the name of the elementary or structure data type prefixed with the letter P to denote a pointer. The attributes of the pointer are defined by the MQPOINTER macro variable; the value of this macro variable depends on the environment. The following illustrates how to declare pointer data types:

```
#define MQPOINTER                  /* depends on environment */
...
typedef MQLONG  MQPOINTER PMQLONG;  /* pointer to MQLONG */
typedef MQMD   MQPOINTER PMQMD;     /* pointer to MQMD */
```

Manipulating binary strings:

Strings of binary data are declared as one of the MQBYTEn data types.

Whenever you copy, compare, or set fields of this type, use the C functions `memcpy`, `memcmp`, or `memset`:

```
#include <string.h>
#include "cmqc.h"

MQMD MyMsgDesc;

memcpy(MyMsgDesc.MsgId,           /* set "MsgId" field to nulls    */
       MQMI_NONE,                 /* ...using named constant       */
       sizeof(MyMsgDesc.MsgId));

memset(MyMsgDesc.CorrelId,        /* set "CorrelId" field to nulls */
       0x00,                      /* ...using a different method   */
       sizeof(MQBYTE24));
```

Do not use the string functions strcpy, strcmp, strncpy, or strncmp because these do not work correctly with data declared as MQBYTE24.

Manipulating character strings:

When the queue manager returns character data to the application, the queue manager always pads the character data with blanks to the defined length of the field. The queue manager *does not* return null-terminated strings, but you can use them in your input. Therefore, when copying, comparing, or concatenating such strings, use the string functions strncpy, strncmp, or strncat.

Do not use the string functions that require the string to be terminated by a null (strcpy, strcmp, and strcat). Also, do not use the function strlen to determine the length of the string; use instead the sizeof function to determine the length of the field.

Initial values for structures:

The include file <cmqc.h> defines various macro variables that you can use to provide initial values for the structures when declaring instances of those structures. These macro variables have names of the form MQxxx_DEFAULT, where MQxxx represents the name of the structure. Use them like this:

```
MQMD   MyMsgDesc = {MQMD_DEFAULT};
MQPMO  MyPutOpts = {MQPMO_DEFAULT};
```

For some character fields, the MQI defines particular values that are valid (for example, for the *StrucId* fields or for the *Format* field in MQMD). For each of the valid values, two macro variables are provided:

- One macro variable defines the value as a string whose length, excluding the implied null, matches exactly the defined length of the field. For example, (the symbol ␣ represents a blank character):

  ```
  #define MQMD_STRUC_ID "MD␣␣"
  #define MQFMT_STRING "MQSTR␣␣␣"
  ```

  Use this form with the memcpy and memcmp functions.
- The other macro variable defines the value as an array of char; the name of this macro variable is the name of the string form suffixed with _ARRAY. For example:

  ```
  #define MQMD_STRUC_ID_ARRAY 'M','D','␣','␣'
  #define MQFMT_STRING_ARRAY 'M','Q','S','T','R','␣','␣','␣'
  ```

  Use this form to initialize the field when an instance of the structure is declared with values different from those provided by the MQMD_DEFAULT macro variable.

Initial values for dynamic structures:

When a variable number of instances of a structure are required, the instances are usually created in main storage obtained dynamically using the calloc or malloc functions.

To initialize the fields in such structures, the following technique is recommended:

1. Declare an instance of the structure using the appropriate MQxxx_DEFAULT macro variable to initialize the structure. This instance becomes the *model* for other instances:

   ```
   MQMD ModelMsgDesc = {MQMD_DEFAULT};
                                    /* declare model instance */
   ```

Code the static or auto keywords on the declaration to give the model instance static or dynamic lifetime, as required.

2. Use the calloc or malloc functions to obtain storage for a dynamic instance of the structure:

```
PMQMD InstancePtr;
InstancePtr = malloc(sizeof(MQMD));
                                    /* get storage for dynamic instance */
```

3. Use the memcpy function to copy the model instance to the dynamic instance:

```
memcpy(InstancePtr,&ModelMsgDesc,sizeof(MQMD));
                                    /* initialize dynamic instance */
```

Use from C++:

For the C++ programming language, the header files contain the following additional statements that are included only when a C++ compiler is used:

```
#ifdef __cplusplus
  extern "C" {
#endif

/* rest of header file */

#ifdef __cplusplus
  }
#endif
```

Coding in COBOL

Note the information in the following sections when coding WebSphere MQ programs in COBOL.

Named constants:

In this book, the names of constants are shown containing the underscore character (_) as part of the name. In COBOL, you must use the hyphen character (-) in place of the underscore.

Constants that have character-string values use the single quotation mark character (') as the string delimiter. To make the compiler accept this character, use the compiler option APOST.

The copy file CMQV contains declarations of the named constants as level-10 items. To use the constants, declare the level-01 item explicitly, then use the COPY statement to copy in the declarations of the constants:

```
 WORKING-STORAGE SECTION.
 01  MQM-CONSTANTS.
     COPY CMQV.
```

However, this method causes the constants to occupy storage in the program even if they are not referred to. If the constants are included in many separate programs within the same run unit, multiple copies of the constants will exist; this might result in a significant amount of main storage being used. You can avoid this by adding the GLOBAL clause to the level-01 declaration:

```
* Declare a global structure to hold the constants
 01  MQM-CONSTANTS GLOBAL.
     COPY CMQV.
```

This allocates storage for only *one* set of constants within the run unit; the constants, however, can be referred to by *any* program within the run unit, not just the program that contains the level-01 declaration.

Coding in System/390 assembler language

Note the information in the following sections when coding WebSphere MQ for z/OS programs in assembler language.

Names:

In this book, the names of parameters in the descriptions of calls, and the names of fields in the descriptions of structures are shown in mixed case. In the assembler-language macros supplied with WebSphere MQ, all names are in uppercase.

Using the MQI calls:

The MQI is a call interface, so assembler-language programs must observe the OS linkage convention.

In particular, before they issue an MQI call, assembler-language programs must point register R13 at a save area of at least 18 full words. This save area provides storage for the called program. It stores the registers of the caller before their contents are destroyed, and restores the contents of the caller's registers on return.

Note: This is important for CICS assembler-language programs that use the DFHEIENT macro to set up their dynamic storage, but that choose to override the default DATAREG from R13 to other registers. When the CICS Resource Manager Interface receives control from the stub, it saves the current contents of the registers at the address to which R13 is pointing. Failing to reserve a proper save area for this purpose gives unpredictable results, and will probably cause an abend in CICS.

Declaring constants:

Most constants are declared as equates in macro CMQA.

However, the following constants cannot be defined as equates, and these are not included when you call the macro using default options:

- MQACT_NONE
- MQCI_NONE
- MQFMT_NONE
- MQFMT_ADMIN
- MQFMT_COMMAND_1
- MQFMT_COMMAND_2
- MQFMT_DEAD_LETTER_HEADER
- MQFMT_EVENT
- MQFMT_IMS
- MQFMT_IMS_VAR_STRING
- MQFMT_PCF
- MQFMT_STRING
- MQFMT_TRIGGER
- MQFMT_XMIT_Q_HEADER

- MQMI_NONE

To include them, add the keyword EQUONLY=NO when you call the macro.

CMQA is protected against multiple declaration, so you can include it many times. However, the keyword EQUONLY takes effect only the first time that the macro is included.

Specifying the name of a structure:

To allow more than one instance of a structure to be declared, the macro that generates the structure prefixes the name of each field with a user-specifiable string and an underscore character (_).

Specify the string when you invoke the macro. If you do not specify a string, the macro uses the name of the structure to construct the prefix:

```
* Declare two object descriptors
         CMQODA                Prefix used="MQOD_" (the default)
MY_MQOD  CMQODA                Prefix used="MY_MQOD_"
```

The structure declarations in the *WebSphere MQ Application Programming Reference* show the default prefix.

Specifying the form of a structure:

The macros can generate structure declarations in one of two forms, controlled by the DSECT parameter:

DSECT=YES
: An assembler-language DSECT instruction is used to start a new data section; the structure definition immediately follows the DSECT statement. No storage is allocated, so no initialization is possible. The label on the macro invocation is used as the name of the data section; if no label is specified, the name of the structure is used.

DSECT=NO
: Assembler-language DC instructions are used to define the structure at the current position in the routine. The fields are initialized with values, which you can specify by coding the relevant parameters on the macro invocation. Fields for which no values are specified on the macro invocation are initialized with default values.

  DSECT=NO is assumed if the DSECT parameter is not specified.

Controlling the listing:

You can control the appearance of the structure declaration in the assembler-language listing by means of the LIST parameter:

LIST=YES
: The structure declaration appears in the assembler-language listing.

LIST=NO
: The structure declaration does not appear in the assembler-language listing. This is assumed if the LIST parameter is not specified.

Specifying initial values for fields:

You can specify the value to be used to initialize a field in a structure by coding the name of that field (without the prefix) as a parameter on the macro invocation, accompanied by the value required.

For example, to declare a message descriptor structure with the *MsgType* field initialized with MQMT_REQUEST, and the *ReplyToQ* field initialized with the string MY_REPLY_TO_QUEUE, use the following code:

```
MY_MQMD        CMQMDA           MSGTYPE=MQMT_REQUEST,             X
               REPLYTOQ=MY_REPLY_TO_QUEUE
```

If you specify a named constant (or equate) as a value on the macro invocation, use the CMQA macro to define the named constant. You must not enclose in single quotation marks (' ') values that are character strings.

Writing reenterable programs:

WebSphere MQ uses its structures for both input and output. If you want your program to remain reenterable:

1. Define working storage versions of the structures as DSECTs, or define the structures inline within an already-defined DSECT. Then copy the DSECT to storage that is obtained using:
   - For batch and TSO programs, the STORAGE or GETMAIN z/OS assembler macros
   - For CICS, the working storage DSECT (DFHEISTG) or the EXEC CICS GETMAIN command

   To correctly initialize these working storage structures, copy a constant version of the corresponding structure to the working storage version.

   Note: The MQMD and MQXQH structures are each more than 256 bytes long. To copy these structures to storage, use the MVCL assembler instruction.
2. Reserve space in storage by using the LIST form (`MF=L`) of the CALL macro. When you use the CALL macro to make an MQI call, use the EXECUTE form (`MF=E`) of the macro, using the storage reserved earlier, as shown in the example under "Using CEDF." For more examples of how to do this, see the assembler language sample programs as shipped with WebSphere MQ.

Use the assembler language RENT option to help you to determine if your program is reenterable.

For information on writing reenterable programs, see *MVS/ESA Application Development Guide: Assembler Language Programs*, GC28-1644.

Using CEDF:

If you want to use the CICS-supplied transaction, CEDF (CICS Execution Diagnostic Facility) to help you to debug your program, add the `,VL` keyword to each `CALL` statement, for example:

```
CALL MQCONN,(NAME,HCONN,COMPCODE,REASON),MF=(E,PARMAREA),VL
```

The above example is reenterable assembler-language code where `PARMAREA` is an area in the working storage that you specified.

Coding in RPG

Supported only on WebSphere MQ for i5/OS.

In this book, the parameters of calls, the names of data types, the fields of structures, and the names of constants are described using their long names. In RPG, these names are abbreviated to six or fewer uppercase characters. For

example, the field *MsgType* becomes *MDMT* in RPG. For more information, see the *WebSphere MQ for i5/OS Application Programming Reference*.

Coding in PL/I

PL/I is supported on z/OS only.

Note the information in the following sections when coding WebSphere MQ for z/OS programs in PL/I.

Structures:

Structures are declared with the BASED attribute, and so do not occupy any storage unless the program declares one or more instances of a structure.

An instance of a structure can be declared using the `like` attribute, for example:

```
dcl my_mqmd       like MQMD;  /* one instance */
dcl my_other_mqmd like MQMD;  /* another one */
```

The structure fields are declared with the INITIAL attribute; when the `like` attribute is used to declare an instance of a structure, that instance inherits the initial values defined for that structure. You need to set only those fields where the value required is different from the initial value.

PL/I is not sensitive to case, and so the names of calls, structure fields, and constants can be coded in lowercase, uppercase, or mixed case.

Named constants:

The named constants are declared as macro variables; as a result, named constants that are not referred to by the program do not occupy any storage in the compiled procedure.

However, the compiler option that causes the source to be processed by the macro preprocessor must be specified when the program is compiled.

All the macro variables are character variables, even the ones that represent numeric values. Although this might seem counter intuitive, it does not result in any data-type conflict after the macro variables have been substituted by the macro processor, for example:

```
%dcl MQMD_STRUC_ID char;
%MQMD_STRUC_ID = '''MD  ''';

%dcl MQMD_VERSION_1 char;
%MQMD_VERSION_1 = '1';
```

Coding in Visual Basic

Note: Outside the .NET environment, support for Visual Basic (VB) in WebSphere MQ has been stabilized at the V6.0 level. Most new function added to WebSphere MQ 7.0 or later is not available to VB applications. If you are programming in VB.NET, use the WebSphere MQ .NET classes. For more information, see *WebSphere MQ Using .NET*.

Visual Basic is supported only on Windows.

To avoid unintended translation of binary data passing between Visual Basic and WebSphere MQ, use an MQBYTE definition instead of MQSTRING. CMQB.BAS defines several new MQBYTE types that are equivalent to a C byte definition and uses these within WebSphere MQ structures. For example, for the MQMD (message descriptor) structure, MsgId (message identifier) is defined as MQBYTE24.

Visual Basic does not have a pointer datatype, so references to other WebSphere MQ data structures are by offset rather than pointer. Declare a compound structure consisting of the two component structures, and specify the compound structure on the call. WebSphere MQ support for Visual Basic provides an MQCONNXAny call to make this possible and allow client applications to specify the channel properties on a client connection. It accepts an untyped structure (MQCNOCD) in place of the usual MQCNO structure.

The MQCNOCD structure is a compound structure consisting of an MQCNO followed by an MQCD. This structure is declared in the exits header file CMQXB. Use the routine MQCNOCD_DEFAULTS to initialize an MQCNOCD structure. A sample making MQCONNX calls is provided (amqscnxb.vbp).

MQCONNXAny has the same parameters as MQCONNX, except that the *ConnectOpts* parameter is declared as being of Any datatype rather than of MQCNO datatype. This allows the function to accept either the MQCNO or the MQCNOCD structure. This function is declared in the main header file CMQB.

z/OS batch considerations

z/OS batch programs that call the MQI can be in either supervisor or problem state.

However, they must meet the following conditions:

- They must be in task mode, not service request block (SRB) mode.
- They must be in Primary address space control (ASC) mode (not Access Register ASC mode).
- They must not be in cross-memory mode. The primary address space number (ASN) must be equal to the secondary ASN and the home ASN.
- They must not be used as MPF exit programs.
- No z/OS locks can be held.
- There can be no function recovery routines (FRRs) on the FRR stack.
- Any program status word (PSW) key can be in force for the MQCONN or MQCONNX call (provided the key is compatible with using storage that is in the TCB key), but subsequent calls that use the connection handle returned by MQCONN or MQCONNX:
  - Must have the same PSW key that was used on the MQCONN or MQCONNX call
  - Must have parameters accessible (for write, where appropriate) under the same PSW key
  - Must be issued under the same task (TCB), but not in any subtask of the task
- They can be in either 24-bit or 31-bit addressing mode. However, if 24-bit addressing mode is in force, parameter addresses must be interpreted as valid 31-bit addresses.

If any of these conditions is not met, a program check might occur. In some cases the call will fail and a reason code will be returned.

UNIX signal handling

This section does not apply to WebSphere MQ for z/OS or WebSphere MQ for Windows.

In general, UNIX and i5/OS systems have moved from a nonthreaded (process) environment to a multithreaded environment. In the nonthreaded environment, some functions could be implemented only by using signals, though most applications did not need to be aware of signals and signal handling. In the multithreaded environment, thread-based primitives support some of the functions that used to be implemented in the nonthreaded environments using signals.

In many instances, signals and signal handling, although supported, do not fit well into the multithreaded environment and various restrictions exist. This can be particularly problematic when you are integrating application code with different middleware libraries (running as part of the application) in a multithreaded environment where each is trying to handle signals. The traditional approach of saving and restoring signal handlers (defined per process), which worked when there was only one thread of execution within a process, does not work in a multithreaded environment. This is because many threads of execution could be trying to save and restore a process-wide resource, with unpredictable results.

Unthreaded applications

(Not applicable on Solaris as all applications are considered threaded even if they use only a single thread.)

Each MQI function sets up its own signal handler for the signals:
SIGALRM
SIGBUS
SIGFPE
SIGSEGV
SIGILL

Users' handlers for these are replaced for the duration of the MQI function call. Other signals can be caught in the normal way by user-written handlers. If you do not install a handler, the default actions (for example, ignore, core dump, or exit) are left in place.

After WebSphere MQ handles a synchronous signal (SIGSEGV, SIGBUS, SIGFPE, SIGILL), it attempts to pass the signal to any registered signal handler before making the MQI function call.

Threaded applications

A thread is considered to be connected to WebSphere MQ from MQCONN (or MQCONNX) until MQDISC.

Synchronous signals:

Synchronous signals arise in a specific thread.

UNIX safely allows the setting up of a signal handler for such signals for the whole process. However, WebSphere MQ sets up its own handler for the following signals, in the application process, while any thread is connected to WebSphere MQ:
SIGBUS

SIGFPE
SIGSEGV
SIGILL

If you are writing multithreaded applications, there is only one process-wide signal handler for each signal. When WebSphere MQ sets up its own synchronous signal handlers it saves any previously registered handlers for each signal. After WebSphere MQ handles one of the signals listed above, WebSphere MQ attempts to call the signal handler that was in effect at the time of the first WebSphere MQ connection within the process. The previously-registered handlers are restored when all application threads have disconnected from WebSphere MQ.

Because signal handlers are saved and restored by WebSphere MQ, application threads ideally should not establish signal handlers for these signals while there is any possibility that another thread of the same process is also connected to WebSphere MQ.

Note: When an application, or a middleware library (running as part of an application), establishes a signal handler while a thread is connected to WebSphere MQ, the application's signal handler must call the corresponding WebSphere MQ handler during the processing of that signal.

When establishing and restoring signal handlers, the general principle is that the last signal handler to be saved must be the first to be restored:

- When an application establishes a signal handler after connecting to WebSphere MQ, the previous signal handler must be restored before the application disconnects from WebSphere MQ.
- When an application establishes a signal handler before connecting to WebSphere MQ, the application must disconnect from WebSphere MQ before restoring its signal handler.

Note: Failure to observe the general principle that the last signal handler to be saved must be the first to be restored can result in unexpected signal handling in the application and, potentially, the loss of signals by the application.

Asynchronous signals:

WebSphere MQ does not use any asynchronous signals in threaded applications unless they are client applications.

Additional considerations for threaded client applications:

WebSphere MQ handles the following signals during I/O to a server. These signals are defined by the communications stack. The application must not establish a signal handler for these signals while a thread is connected to a queue manager:
SIGPIPE (for TCP/IP)

Additional considerations

Fastpath (trusted) applications:

Fastpath applications run in the same process as WebSphere MQ and so are running in the multithreaded environment.

In this environment WebSphere MQ handles the synchronous signals SIGSEGV, SIGBUS, SIGFPE, and SIGILL. All other signals must not be delivered to the

Fastpath application while it is connected to WebSphere MQ. Instead they must be blocked or handled by the application. If a Fastpath application intercepts such an event, the queue manager must be stopped and restarted, or it may be left in an undefined state. For a full list of the restrictions for Fastpath applications under MQCONNX see "Connecting to a queue manager using the MQCONNX call" on page 94.

MQI function calls within signal handlers:

While you are in a signal handler, do not call an MQI function.

If you try to call an MQI function from a signal handler while another MQI function is active, MQRC_CALL_IN_PROGRESS is returned. If you try to call an MQI function from a signal handler while no other MQI function is active, it is likely to fail sometime during the operation because of the operating system restrictions where only selective calls can be issued from, or within, a handler.

In the case of C++ destructor methods, which may be called automatically during program exit, you might not be able to stop the MQI functions from being called. Ignore any errors about MQRC_CALL_IN_PROGRESS. If a signal handler calls exit(), WebSphere MQ backs out uncommitted messages in syncpoint as usual and closes any open queues.

Signals during MQI calls:

MQI functions do not return the code EINTR or any equivalent to application programs.

If a signal occurs during an MQI call, and the handler calls *return*, the call continues to run as if the signal had not happened. In particular, MQGET cannot be interrupted by a signal to return control immediately to the application. If you want to break out of an MQGET, set the queue to GET_DISABLED; alternatively, use a loop around a call to MQGET with a finite time expiry (MQGMO_WAIT with gmo.WaitInterval set), and use your signal handler (in a nonthreaded environment) or equivalent function in a threaded environment to set a flag which breaks the loop.

In the AIX environment, WebSphere MQ requires that system calls interrupted by signals are restarted. When establishing your own signal handler with sigaction(2), set the SA_RESTART flag in the sa_flags field of the new action structure otherwise WebSphere MQ might be unable to complete any call interrupted by a signal.

User exits and installable services:

User exits and installable services that run as part of a WebSphere MQ process in a multithreaded environment have the same restrictions as for fastpath applications. They should be considered as permanently connected to WebSphere MQ and so do not use signals or non-threadsafe operating system calls.

VMS exit handlers:

Users can install exit handlers for an MQ application using the SYS$DCLEXH system service.

The exit handler receives control when an image exits. An image exit will normally occur when you call the Exit ($EXIT) or Force Exit ($FORCEX) service. The

$FORCEX will interrupt the target process in user mode. Then all user-mode exit handlers (established via $DCLEXH) will begin to execute in reverse order of establishment. For more details on exit handlers and $FORCEX, please refer to the *VMS Programming Concepts Manual* and the *VMS System Services Manual*.

If you call an MQI function from within an exit handler, the behavior of the function depends on the way the image was terminated. If the image was terminated while another MQI function is active, an `MQRC_CALL_IN_PROGRESS` will be returned.

It is possible to call an MQI function from within an exit handler if no other MQI function is active and upcalls are disabled for the MQ application. If upcalls are enabled for the MQ application, it will fail with the reason code `MQRC_HCONN_ERROR`.

The scope of an MQCONN or MQCONNX call is usually the thread that issued it. If upcalls are enabled, the exit handler will be run as a separate thread and the connection handles can not be shared.

Exit handlers are invoked within the interrupted context of the target process. It is up to the application to ensure that actions taken by a handler are safe and reliable, for the asynchronously interrupted context they are called from.

Connecting to and disconnecting from a queue manager

To use WebSphere MQ programming services, a program must have a connection to a queue manager.

The way that this connection is made depends on the platform and the environment in which the program is operating:

z/OS batch, WebSphere MQ for i5/OS, WebSphere MQ on UNIX systems, and WebSphere MQ for Windows

Programs that run in these environments can use the MQCONN MQI call to connect to, and the MQDISC call to disconnect from, a queue manager. Alternatively, programs can use the MQCONNX call. This chapter describes how to use these calls.

z/OS batch programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

IMS The IMS control region is connected to one or more queue managers when it starts. This connection is controlled by IMS commands. (For information on how to control the IMS adapter of WebSphere MQ for z/OS, see the *WebSphere MQ for z/OS System Administration Guide*.) However, writers of message queuing IMS programs must use the MQCONN MQI call to specify the queue manager to which they want to connect. They can use the MQDISC call to disconnect from that queue manager. This chapter describes how writers of such programs should use these calls.

Before the IMS adapter processes a message for another user following a Get Unique call from the IOPCB, or one implied by a checkpoint call, the adapter ensures that the application closes handles and disconnects from the queue manager.

IMS programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

CICS Transaction Server for OS/390® and CICS for MVS/ESA™

CICS programs do not need to do any work to connect to a queue

manager because the CICS system itself is connected. This connection is usually made automatically at initialization, but you can also use the CKQC transaction, which is supplied with WebSphere MQ for z/OS. CKQC is discussed in the *WebSphere MQ for z/OS System Administration Guide*.

CICS tasks can connect only to the queue manager to which the CICS region, itself, is connected.

Note: CICS programs can also use the MQI connect and disconnect calls (MQCONN and MQDISC). You might want to do this so that you can port these applications to non-CICS environments with a minimum of recoding. However, these calls *always* complete successfully in a CICS environment. This means that the return code might not reflect the true state of the connection to the queue manager.

TXSeries for Windows and Open Systems

These programs do not need to do any work to connect to a queue manager because the CICS system itself is connected. Therefore, only one connection at a time is supported. CICS applications must issue an MQCONN call to obtain a connection handle, and an MQDISC call before they exit.

This chapter introduces connecting to and disconnecting from a queue manager, under these headings:

- "Connecting to a queue manager using the MQCONN call"
- "Connecting to a queue manager using the MQCONNX call" on page 94
- "Disconnecting programs from a queue manager using MQDISC" on page 98

Connecting to a queue manager using the MQCONN call

In general, you can connect either to a specific queue manager, or to the default queue manager:

- For WebSphere MQ for z/OS, in the batch environment, the default queue manager is specified in the CSQBDEFV module.
- For WebSphere MQ for i5/OS, and WebSphere MQ on UNIX systems, the default queue manager is specified in the mqs.ini file.
- For WebSphere MQ for Windows, the default queue manager is specified in the registry.

Alternatively, in the z/OS MVS™ batch, TSO, and RRS environments you can connect to any one queue manager within a queue-sharing group. The MQCONN or MQCONNX request selects any one of the active members of the group.

The queue manager that you connect to must be *local* to the task. This means that it must belong to the same system as the WebSphere MQ application.

In the IMS environment, the queue manager must be connected to the IMS control region and to the dependent region that the program uses. The default queue manager is specified in the CSQQDEFV module when WebSphere MQ for z/OS is installed.

With the TXSeries CICS environment, and TXSeries for Windows and AIX, the queue manager must be defined as an XA resource to CICS.

To connect to the default queue manager, call MQCONN, specifying a name consisting entirely of blanks or starting with a null (X'00') character.

Within WebSphere MQ on UNIX systems, an application must be authorized for it to successfully connect to a queue manager. For more information, see the the *WebSphere MQ System Administration Guide*.

The output from MQCONN is:

- A connection handle (Hconn)
- A completion code
- A reason code

Use the connection handle on subsequent MQI calls.

If the reason code indicates that the application is already connected to that queue manager, the connection handle that is returned is the same as the one that was returned when the application first connected. The application should not issue the MQDISC call in this situation because the calling application will expect to remain connected.

The scope of the connection handle is the same as that of the object handle (see "Opening objects using the MQOPEN call" on page 100).

Descriptions of the parameters are given in the description of the MQCONN call in the *WebSphere MQ Application Programming Reference*.

The MQCONN call fails if the queue manager is in a quiescing state when you issue the call, or if the queue manager is shutting down.

Scope of MQCONN or MQCONNX

Within WebSphere MQ for i5/OS, WebSphere MQ on UNIX systems, and WebSphere MQ for Windows, the scope of an MQCONN or MQCONNX call is usually the thread that issued it.

That is, the connection handle returned from the call is valid only within the thread that issued the call. Only one call can be made at any one time using the handle. If it is used from a different thread, it is rejected as invalid. If you have multiple threads in your application and each wants to use WebSphere MQ calls, each one must issue MQCONN or MQCONNX. Alternatively, consider "Shared (thread independent) connections with MQCONNX" on page 96.[1]

On WebSphere MQ for i5/OS, WebSphere MQ on UNIX systems, and WebSphere MQ for Windows, each thread in an application can connect to different queue managers; on other systems, all concurrent connections within a process must be to the same queue manager.

If your application is running as a client, it can connect to more than one queue manager within a thread.

1. When using multithreaded applications with WebSphere MQ on UNIX systems you need to ensure that the applications have a sufficient stack size for the threads. You are recommended to use a stack size of 256KB, or larger, when multithreaded applications are making MQI calls, either by themselves or, with other signal handlers (for example, CICS).

Connecting to a queue manager using the MQCONNX call

The MQCONNX call is similar to the MQCONN call, but includes options to control the way that the call works.

As input to MQCONNX, you can supply a queue manager name, or a queue-sharing group name on z/OS shared queue systems. The output from MQCONNX is:

- A connection handle (Hconn)
- A completion code
- A reason code

You use the connection handle on subsequent MQI calls.

A description of all the parameters of MQCONNX is given in the *WebSphere MQ Application Programming Reference*. The `Options` field allows you to set STANDARD_BINDING, FASTPATH_BINDING, SHARED_BINDING, or ISOLATED_BINDING for any version of MQCNO. You can also make shared (thread independent) connections using a MQCONNX call. See "Shared (thread independent) connections with MQCONNX" on page 96 for more information about these.

MQCNO_STANDARD_BINDING

By default, MQCONNX (like MQCONN) implies two logical threads where the WebSphere MQ application and the local queue manager agent run in separate processes. The WebSphere MQ application requests the WebSphere MQ operation and the local queue manager agent services the request. This is defined by the MQCNO_STANDARD_BINDING option on the MQCONNX call.

Note: This default maintains the integrity of the queue manager (that is, it makes the queue manager immune to errant programs), but impairs the performance of the MQI calls.

MQCNO_FASTPATH_BINDING

Trusted applications imply that the WebSphere MQ application and the local queue manager agent become the same process. Because the agent process no longer needs to use an interface to access the queue manager, these applications become an extension of the queue manager. This is defined by the MQCNO_FASTPATH_BINDING option on the MQCONNX call.

You need to link trusted applications to the threaded WebSphere MQ libraries. For instructions on how to set up a WebSphere MQ application to run as trusted, see the *WebSphere MQ Application Programming Reference*.

Note: This option compromises the integrity of the queue manager: there is no protection from overwriting its storage. This also applies if the application contains errors that can be exposed to messages and other data in the queue manager too. Consider these issues before using this option.

MQCNO_SHARED_BINDING

Specify this option to make the application and the local queue manager agent run in separate processes. This maintains the integrity of the queue manager, that is, it protects the queue manager from errant programs. However, the application and the local-queue-manager agent share some resources.

MQCNO_SHARED_BINDING is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

MQCNO_ISOLATED_BINDING

Specify this option to make the application and the local queue manager agent run in separate processes, as for MQCNO_SHARED_BINDING. In this case, however, the application process and the local-queue-manager agent are isolated from each other in that they do not share resources.

MQCNO_ISOLATED_BINDING is ignored if the queue manager does not support this type of binding. Processing continues as though the option had not been specified.

On z/OS these options are tolerated, but only a standard bound connection is performed. MQCNO Version 3, for z/OS, allows four alternative options:

MQCNO_SERIALIZE_CONN_TAG_QSG

This allows an application to request that only one instance of an application runs at any one time in a queue-sharing group. This is achieved by registering the use of a connection tag, whose value is specified or derived by the application. The tag is a 128 byte character string specified in the Version 3 MQCNO.

MQCNO_RESTRICT_CONN_TAG_QSG

This is used where an application consists of more than one process (or a TCB), each of which can connect to a queue manager. Connection is permitted only if there is no current use of the tag, or the requesting application is within the same processing scope. This is MVS address space within the same queue-sharing group as the tag owner.

MQCNO_SERIALIZE_CONN_TAG_Q_MGR

This is similar to MQCNO_SERIALIZE_CONN_TAG_QSG, but only the local queue manager is interrogated to see if the requested tag is already in use.

MQCNO_RESTRICT_CONN_TAG_Q_MGR

This is similar to MQCNO_RESTRICT_CONN_TAG_QSG, but only the local queue manager is interrogated to see if the requested tag is already in use.

Restrictions for trusted applications

The following restrictions apply to trusted applications:

- You must explicitly disconnect trusted applications from the queue manager.
- You must stop trusted applications before ending the queue manager with the endmqm command.
- You must not use asynchronous signals and timer interrupts (such as `sigkill`) with MQCNO_FASTPATH_BINDING.
- On all platforms, a thread within a trusted application cannot connect to a queue manager while another thread in the same process is connected to a different queue manager.
- On WebSphere MQ on UNIX systems you must use mqm as the effective userID and groupID for all MQI calls. You can change these IDs before making a

non-MQI call requiring authentication (for example, opening a file), but you *must* change it back to mqm before making the next MQI call.

- On WebSphere MQ for i5/OS:
  1. Trusted applications must run under the QMQM user profile. It is not sufficient that the user profile be a member of the QMQM group or that the program adopt QMQM authority. It might not be possible for the QMQM user profile to be used to sign on to interactive jobs, or to be specified in the job description for jobs running trusted applications. In this case one approach is to use the i5/OS profile swapping API functions, QSYGETPH, QWTSETP, and QSYRLSPH to temporarily change the current user of the job to QMQM while the MQ programs run. Details of these functions, together with an example of their use, is provided in the Security APIs section of the *i5/OS System API Reference*.
  2. Do not cancel trusted applications using System-Request Option 2, or by ending the jobs in which they are running using ENDJOB.
- On WebSphere MQ for HP-UX, multithreaded fast-path applications are likely to need to set a larger stack size than the default. Use a size of 256 KB.
- On WebSphere MQ for Windows trusted 64-bit applications are not supported. If you try to run a trusted 64-bit application, it will be downgraded to a standard bound connection.
- On WebSphere MQ on UNIX systems trusted 32-bit applications are not supported. If you try to run a trusted 32-bit application, it will be downgraded to a standard bound connection.

Shared (thread independent) connections with MQCONNX

Not supported on WebSphere MQ for z/OS.

On WebSphere MQ platforms other than WebSphere MQ for z/OS, a connection made with MQCONN is available only to the thread that made the connection. Options on the MQCONNX call allow you to create a connection that can be shared by all the threads in a process.

Use one of the following options to make a thread independent or shared connection:

MQCNO_HANDLE_SHARE_BLOCK	Creates a shared connection on which, if the connection is currently in use by another thread, an MQI call waits until the current MQI call has completed
MQCNO_HANDLE_SHARE_NO_BLOCK	Creates a shared connection on which, if the connection is currently in use by another thread, an MQI call fails immediately with a reason of MQRC_CALL_IN_PROGRESS
MQCNO_HANDLE_SHARE_NONE	Creates a standard non-shared connection

In the normal MQI environment the default value is MQCNO_HANDLE_SHARE_NONE. In the MTS environment the default value is MQCNO_HANDLE_SHARE_BLOCK.

A connection handle (Hconn) is returned from the MQCONNX call in the usual way. This can be used by subsequent MQI calls from any thread in the process, associating those calls with the Hconn returned from the MQCONNX. MQI calls using a single shared Hconn are serialized across threads.

For example, the following sequence of activity is possible with a shared Hconn:

1. Thread 1 issues MQCONNX and gets a shared Hconn *h1*
2. Thread 1 opens a queue and issues a get request using *h1*
3. Thread 2 issues a put request using *h1*
4. Thread 3 issues a put request using *h1*
5. Thread 2 issues MQDISC using *h1*

While the Hconn is in use by any thread, access to the connection is unavailable to other threads. In circumstances where it is acceptable that a thread waits for any previous call from another thread to complete, use MQCONNX with the option MQCNO_HANDLE_SHARE_BLOCK.

However this can cause difficulties. Suppose that in step 2 above, the thread issues a get request that waits for messages that might not have yet arrived (a get with wait). In this case, threads 2 and 3 are also left waiting (blocked) for as long as the get request takes. If you prefer that your application is notified of calls that are already running on the Hconn, use MQCONNX with the option MQCNO_HANDLE_SHARE_NO_BLOCK.

Shared connection usage notes:

1. Any object handles (Hobj) created by opening an object are associated with an Hconn; so for a shared Hconn, the Hobjs are also shared and usable by any thread using the Hconn. Similarly, any unit of work started under an Hconn is associated with that Hconn; so this too is shared across threads with the shared Hconn.
2. *Any* thread can call MQDISC to disconnect a shared Hconn, not just the thread that called the corresponding MQCONNX. The MQDISC terminates the Hconn making it unavailable to all threads.
3. A single thread can use multiple shared Hconns serially, for example use MQPUT to put one message under one shared Hconn then put another message using another shared Hconn, with each operation being under a different local unit of work.
4. Shared Hconns cannot be used within a global unit of work.

MQCONNX environment variable

On WebSphere MQ for i5/OS, WebSphere MQ for Windows, and WebSphere MQ on UNIX systems, you can use the environment variable, MQ_CONNECT_TYPE in combination with the type of binding specified in the *Options* field. This environment variable allows you to execute the application with the STANDARD_BINDING if any problems occur with the FASTPATH_BINDING. You specify the environment variable with the value FASTPATH or STANDARD to select the type of binding required. However, the FASTPATH binding is used only if the connect option is appropriately specified as shown in Table 4:

Table 4. The MQ_CONNECT_TYPE environment variable

MQCONNX call option	MQ_CONNECT_TYPE environment variable	Result
STANDARD	UNDEFINED	STANDARD
FASTPATH	UNDEFINED	FASTPATH
STANDARD	STANDARD	STANDARD
FASTPATH	STANDARD	STANDARD

Table 4. The MQ_CONNECT_TYPE environment variable (continued)

MQCONNX call option	MQ_CONNECT_TYPE environment variable	Result
STANDARD	FASTPATH	STANDARD
FASTPATH	FASTPATH	FASTPATH

So, to run a trusted application, either:

1. Specify the MQCNO_FASTPATH_BINDING option on the MQCONNX call and the FASTPATH environment variable, or
2. Specify the MQCNO_FASTPATH_BINDING option on the MQCONNX call and leave the environment variable undefined.

If neither MQCNO_STANDARD_BINDING nor MQCNO_FASTPATH_BINDING is specified, you can use MQCNO_NONE, which defaults to MQCNO_STANDARD_BINDING.

Disconnecting programs from a queue manager using MQDISC

When a program that has connected to a queue manager using the MQCONN or MQCONNX call has finished all interaction with the queue manager, it breaks the connection using the MQDISC call, except:

- On CICS Transaction Server for z/OS applications, where the call is optional unless MQCONNX was used and you want to drop the connection tag before the application ends.
- On WebSphere MQ for i5/OS where, when you sign off from the operating system, an implicit MQDISC call is made.

As input to the MQDISC call, you must supply the connection handle (Hconn) that was returned by MQCONN or MQCONNX when you connected to the queue manager.

Except on CICS on z/OS, after MQDISC is called the connection handle (Hconn) is no longer valid, and you cannot issue any further MQI calls until you call MQCONN or MQCONNX again. MQDISC does an implicit MQCLOSE for any objects that are still open using this handle.

If you use MQCONNX to connect on WebSphere MQ for z/OS, MQDISC also ends the scope of the connection tag established by the MQCONNX. However, in a CICS, IMS, or RRS application, if there is an active unit of recovery associated with a connection tag, the MQDISC is rejected with a reason code of MQRC_CONN_TAG_NOT_RELEASED.

Descriptions of the parameters are given in the description of the MQDISC call in the *WebSphere MQ Application Programming Reference*.

When no MQDISC is issued

A standard, non-shared connection (Hconn) is cleaned up when the creating thread terminates. A shared connection is only implicitly backed out and disconnected when the whole process terminates. If the thread that created the shared Hconn terminates while the Hconn still exists the Hconn is still usable.

Authority checking

The MQCLOSE and MQDISC calls usually perform no authority checking.

In the normal course of events a job that has the authority to open or connect to a WebSphere MQ object closes or disconnect from that object. Even if the authority of a job that has connected to or opened a WebSphere MQ object is revoked, the MQCLOSE and MQDISC calls are accepted.

Opening and closing objects

To perform any of the following operations, you must first *open* the relevant WebSphere MQ object:

- Put messages on a queue
- Get (browse or retrieve) messages from a queue
- Set the attributes of an object
- Inquire about the attributes of any object

Use the MQOPEN call to open the object, using the options of the call to specify what you want to do with the object. The only exception is if you want to put a single message on a queue, then close the queue immediately. In this case, you can bypass the *opening* stage by using the MQPUT1 call (see "Putting one message on a queue using the MQPUT1 call" on page 117).

Before you open an object using the MQOPEN call, you must connect your program to a queue manager. This is explained in detail, for all environments, in "Connecting to and disconnecting from a queue manager" on page 91.

There are four types of WebSphere MQ object that you can open:

- Queue
- Namelist
- Process definition
- Queue manager

You open all these objects in a similar way using the MQOPEN call. For more information about WebSphere MQ objects, see "WebSphere MQ objects" on page 48.

You can open the same object more than once, and each time you get a new object handle. You might want to browse messages on a queue using one handle, and remove messages from the same queue using another handle. This saves using up resources to close and reopen the same object. You can also open a queue for browsing *and* removing messages at the same time.

Moreover, you can open multiple objects with a single MQOPEN and close them using MQCLOSE. See "Distribution lists" on page 119 for information about how to do this.

When you attempt to open an object, the queue manager checks that you are authorized to open that object for the options that you specify in the MQOPEN call.

Objects are closed automatically when a program disconnects from the queue manager. In the IMS environment, disconnection is forced when a program starts

processing for a new user following a GU (get unique) IMS call. On the i5/OS platform, objects are closed automatically when a job ends.

It is good programming practice to close objects you have opened. Use the MQCLOSE call to do this.

This chapter introduces opening and closing WebSphere MQ objects, under these headings:

Opening objects using the MQOPEN call

As input to the MQOPEN call, you must supply:

- A connection handle. For CICS applications on z/OS, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNX call. For other programs, always use the connection handle returned by the MQCONN or MQCONNX call.
- A description of the object that you want to open, using the object descriptor structure (MQOD).
- One or more options that control the action of the call.

The output from MQOPEN is:

- An object handle that represents your access to the object. Use this on input to any subsequent MQI calls.
- A modified object-descriptor structure, if you are creating a dynamic queue (and it is supported on your platform).
- A completion code.
- A reason code.

Scope of an object handle

The scope of an object handle (Hobj) is the same as the scope of a connection handle (Hconn).

This is covered in "Scope of MQCONN or MQCONNX" on page 93 and "Shared (thread independent) connections with MQCONNX" on page 96. However, there are additional considerations in some environments :

CICS In a CICS program, you can use the handle only within the same CICS task from which you made the MQOPEN call.

IMS and z/OS batch
In the IMS and batch environments, you can use the handle within the same task, but not within any subtasks.

Descriptions of the parameters of the MQOPEN call are given in the *WebSphere MQ Application Programming Reference*.

The following sections describe the information that you must supply as input to MQOPEN.

Identifying objects (the MQOD structure)

Use the MQOD structure to identify the object that you want to open. This structure is an input parameter for the MQOPEN call. (The structure is modified by the queue manager when you use the MQOPEN call to create a dynamic queue.)

For full details of the MQOD structure see MQOD - Object descriptor.

For information about using the MQOD structure for distribution lists, see "Using the MQOD structure" on page 120 under "Distribution lists" on page 119.

Name resolution

How the MQOPEN call resolves queue and queue manager names.

Note: A Queue manager alias is a remote queue definition without an `RNAME` field.

When you open a WebSphere MQ queue, the MQOPEN call performs a name resolution function on the queue name that you specify. This determines on which queue the queue manager performs subsequent operations. This means that when you specify the name of an alias queue or a remote queue in your object descriptor (MQOD), the call resolves the name either to a local queue or to a transmission queue. If a queue is opened for any type of input, browse, or set, it resolves to a local queue if there is one, and fails if there is not one. It resolves to a nonlocal queue only if it is opened for output only, inquire only, or output and inquire only. See Table 5 for an overview of the name resolution process. The name that you supply in *ObjectQMgrName* is resolved *before* that in *ObjectName*.

Table 5 also shows how you can use a local definition of a remote queue to define an alias for the name of a queue manager. This allows you to select which transmission queue is used when you put messages on a remote queue, so you could, for example, use a single transmission queue for messages destined for many remote queue managers.

To use the following table, first read down the two left-hand columns, under the heading **Input to MQOD**, and select the appropriate case. Then read across the corresponding row, following any instructions. Following the instructions in the **Resolved names** columns, you can either return to the **Input to MQOD** columns and insert values as directed, or you can exit the table with the results supplied. For example, you might be required to input *ObjectName*.

Table 5. Resolving queue names when using MQOPEN

Input to MQOD		**Resolved names**		
ObjectQMgrName	*ObjectName*	*ObjectQMgrName*	*ObjectName*	**Transmission queue**
Blank or local queue manager	Local queue with no CLUSTER attribute	Local queue manager	Input *ObjectName*	Not applicable (local queue used)
Blank queue manager	Local queue with CLUSTER attribute	Workload management selected cluster queue manager or specific cluster queue manager selected on PUT	Input *ObjectName*	SYSTEM.CLUSTER. TRANSMIT.QUEUE and local queue used SYSTEM.QSG. TRANSMIT.QUEUE (see note)

Table 5. Resolving queue names when using MQOPEN (continued)

Input to MQOD		**Resolved names**		
ObjectQMgrName	*ObjectName*	*ObjectQMgrName*	*ObjectName*	**Transmission queue**
Local queue manager	Local queue with CLUSTER attribute	Local queue manager	Input *ObjectName*	Not applicable (local queue used)
Blank or local queue manager	Model queue	Local queue manager	Generated name	Not applicable (local queue used)
Blank or local queue manager	Alias queue	Perform name resolution again with *ObjectQMgrName* unchanged, and input *ObjectName* set to the *BaseQName* in the alias queue definition object. Must not resolve to an alias queue.		
Local queue manager	Alias queue with CLUSTER attribute	The alias must not resolve to a cluster queue that is not locally defined, or a cluster queue that has the same *ObjectName* as the alias.		
Blank queue manager	Alias queue with CLUSTER attribute	The alias can resolve to a cluster queue with same *ObjectName* as the alias.		
Blank or local queue manager	Local definition of a remote queue	Perform name resolution again with *ObjectQMgrName* set to *RemoteQMgrName*, and *ObjectName* set to *RemoteQName*. Must not resolve remote queues		Name of *XmitQName* attribute, if non-blank; otherwise *RemoteQMgrName* in the remote queue definition object. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Blank queue manager	No matching local object; cluster queue found	Workload management selected cluster queue manager or specific cluster queue manager selected on PUT	Input *ObjectName*	SYSTEM.CLUSTER. TRANSMIT.QUEUE SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Blank or local queue manager	No matching local object; cluster queue not found		Error, queue not found	Not applicable
Name of queue manager in same queue sharing group as local queue manager	Local shared queue	Local queue manager	Input *ObjectName*	Not applicable
Name of a local transmission queue	(Not resolved)	Input *ObjectQMgrName*	Input *ObjectName*	Input *ObjectQMgrName* SYSTEM.QSG. TRANSMIT.QUEUE (see note)

Table 5. Resolving queue names when using MQOPEN (continued)

Input to MQOD		**Resolved names**		
ObjectQMgrName	*ObjectName*	*ObjectQMgrName*	*ObjectName*	**Transmission queue**
Queue manager alias definition (*RemoteQMgrName* may be the local queue manager)	(Not resolved, remote queue)	Perform name resolution again with *ObjectQMgrName* set to *RemoteQMgrName*. Must not resolve to remote queues	Input *ObjectName*	Name of *XmitQName* attribute, if non-blank; otherwise *RemoteQMgrName* in the remote queue definition object. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Queue manager is not the name of any local object; cluster queue managers or queue manager alias found	(Not resolved)	*ObjectQMgrName* or specific cluster queue manager selected on PUT	Input *ObjectName*	SYSTEM.CLUSTER. TRANSMIT.QUEUE SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Queue manager is not the name of any local object; no cluster objects found	(Not resolved)	Input *ObjectQMgrName*	Input *ObjectName*	*DefXmitQName* attribute of the queue manager where *DefXmitQName* is supported. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Note: The SYSTEM.QSG.TRANSMIT.QUEUE is used if local and remote queue managers are in the same queue-sharing group; intra-group queuing is enabled.				

Note:

1. *BaseQName* is the name of the base queue from the definition of the alias queue.
2. *RemoteQName* is the name of the remote queue from the local definition of the remote queue.
3. *RemoteQMgrName* is the name of the remote queue manager from the local definition of the remote queue.
4. *XmitQName* is the name of the transmission queue from the local definition of the remote queue.
5. When using WebSphere MQ for z/OS queue managers that are part of a queue-sharing group (QSG), the name of the QSG can be used instead of the local queue manager name in Table 5 on page 101.
6. In the *ObjectName* column of the table, CLUSTER refers to both the CLUSTER and CLUSNL attributes of the queue.

Opening an alias queue also opens the base queue to which the alias resolves, and opening a remote queue also opens the transmission queue. Therefore you cannot delete either the queue that you specify or the queue to which it resolves while the other one is open.

The resolved queue name and the resolved queue manager name are stored in the *ResolvedQName* and *ResolvedQMgrName* fields in the MQOD.

For more information about name resolution in a distributed queuing environment see *WebSphere MQ Intercommunications*.

Using the options of the MQOPEN call

In the *Options* parameter of the MQOPEN call, you must choose one or more options to control the access that you are given to the object that you are opening. With these options you can:

- Open a queue and specify that all messages put to that queue must be directed to the same instance of it
- Open a queue to allow you to put messages on it
- Open a queue to allow you to browse messages on it
- Open a queue to allow you to remove messages from it
- Open an object to allow you to inquire about and set its attributes (but you can set the attributes of queues only)
- Open a topic or topic string to publish messages to it
- Associate context information with a message
- Nominate an alternate user identifier to be used for security checks
- Control the call if the queue manager is in a quiescing state

MQOPEN option for cluster queue:

To route all messages put to a queue using MQPUT to the same queue manager by the same route, use the MQOO_BIND_ON_OPEN option on the MQOPEN call.

To specify that a destination is to be selected at MQPUT time, that is, on a message-by-message basis, use the MQOO_BIND_NOT_FIXED option on the MQOPEN call. If you specify neither of these options the default, MQOO_BIND_AS_Q_DEF, is used. In this case the binding used for the queue handle is taken from the *DefBind* queue attribute, which can take the value MQBND_BIND_ON_OPEN or MQBND_BIND_NOT_FIXED.

If the queue that you open is not a cluster queue, the MQOO_BIND_* options are ignored. If you specify the name of the local queue manager in the MQOD, the local instance of the cluster queue is selected. If the queue manager name is blank, any instance can be selected. See *WebSphere MQ Queue Manager Clusters* for more information.

MQOPEN option for putting messages:

To open a queue or topic to put messages on it, use the MQOO_OUTPUT option.

MQOPEN option for browsing messages:

To open a queue so that you can *browse* the messages on it, use the MQOPEN call with the MQOO_BROWSE option.

This creates a *browse cursor* that the queue manager uses to identify the next message on the queue. For more information, see “Browsing messages on a queue” on page 156.

Note:

1. You cannot browse messages on a remote queue; do not open a remote queue using the MQOO_BROWSE option.
2. You cannot specify this option when opening a distribution list. For further information about distribution lists, see “Distribution lists” on page 119.

3. Use the MQOO_CO_OP in conjunction with MQOO_BROWSE if you are using cooperative browsing; see Options (MQLONG)

MQOPEN options for removing messages:

Three options control the opening of a queue to remove messages from it.

You can use only one of them in any MQOPEN call. These options define whether your program has exclusive or shared access to the queue. *Exclusive access* means that, until you close the queue, only you can remove messages from it. If another program attempts to open the queue to remove messages, its MQOPEN call fails. *Shared access* means that more than one program can remove messages from the queue.

The most advisable approach is to accept the type of access that was intended for the queue when the queue was defined. The queue definition involved the setting of the *Shareability* and the *DefInputOpenOption* attributes. To accept this access, use the MQOO_INPUT_AS_Q_DEF option. Refer to Table 6 to see how the setting of these attributes affects the type of access that you will be given when you use this option.

Table 6. How queue attributes and options of the MQOPEN call affect access to queues

Queue attributes		**Type of access with MQOPEN options**		
Shareability	*DefInputOpenOption*	**AS_Q_DEF**	**SHARED**	**EXCLUSIVE**
SHAREABLE	SHARED	shared	shared	exclusive
SHAREABLE	EXCLUSIVE	exclusive	shared	exclusive
NOT_SHAREABLE*	SHARED*	exclusive	exclusive	exclusive
NOT_SHAREABLE	EXCLUSIVE	exclusive	exclusive	exclusive
Note: * Although you can define a queue to have this combination of attributes, the default input open option is overridden by the shareability attribute.				

Alternatively:

- If you know that your application can work successfully even if other programs can remove messages from the queue at the same time, use the MQOO_INPUT_SHARED option. Table 6 shows how, in some cases you will be given exclusive access to the queue, even with this option.
- If you know that your application can work successfully only if other programs are prevented from removing messages from the queue at the same time, use the MQOO_INPUT_EXCLUSIVE option.

Note:

1. You cannot remove messages from a remote queue. Therefore you cannot open a remote queue using any of the MQOO_INPUT_* options.
2. You cannot specify this option when opening a distribution list. For further information, see “Distribution lists” on page 119.

MQOPEN options for setting and inquiring about attributes:

To open a queue so that you can set its attributes, use the MQOO_SET option.

You cannot set the attributes of any other type of object (see “Inquiring about and setting object attributes” on page 180).

To open an object so that you can inquire about its attributes, use the MQOO_INQUIRE option.

Note: You cannot specify this option when opening a distribution list.

MQOPEN options relating to message context:

If you want to be able to associate context information with a message when you put it on a queue, you must use one of the message context options when you open the queue.

The options allow you to differentiate between context information that relates to the *user* who originated the message, and that which relates to the *application* that originated the message. Also, you can opt to set the context information when you put the message on the queue, or you can opt to have the context taken automatically from another queue handle.

For more information about the subject of message context, see "Message context" on page 46.

MQOPEN option for alternate user authority:

When you attempt to open an object using the MQOPEN call, the queue manager checks that you have the authority to open that object. If you are not authorized, the call fails.

However, server programs might want the queue manager to check the authorization of the user on whose behalf they are working, rather than the server's own authorization. To do this, they must use the MQOO_ALTERNATE_USER_AUTHORITY option of the MQOPEN call, and specify the alternate user ID in the *AlternateUserId* field of the MQOD structure. Typically, the server would get the user ID from the context information in the message it is processing.

MQOPEN option for queue manager quiescing:

In the CICS environment on z/OS, if you use the MQOPEN call when the queue manager is in a quiescing state, the call always fails.

In other z/OS environments, i5/OS, Windows systems and in UNIX systems environments, the call fails when the queue manager is quiescing only if you use the MQOO_FAIL_IF_QUIESCING option of the MQOPEN call.

MQOPEN option for resolving local queue names:

When you open a local, alias or model queue, the local queue is returned.

However, when you open a remote queue or cluster queue, the *ResolvedQName* and *ResolvedQMgrName* fields of the MQOD structure are filled with the names of the remote queue and remote queue manger found in the remote queue definition, or with the chosen remote cluster queue.

Use the MQOO_RESOLVE_LOCAL_Q option of the MQOPEN call to fill the *ResolvedQName* in the MQOD structure with the name of the local queue that was opened. The *ResolvedQMgrName* is similarly filled with the name of the local queue manager hosting the local queue. This field is available only with Version 3 of the

MQOD structure; if the structure is less than Version 3, MQOO_RESOLVE_LOCAL_Q is ignored without an error being returned.

If you specify MQOO_RESOLVE_LOCAL_Q when opening, for example, a remote queue, *ResolvedQName* is the name of the transmission queue to which messages will be put. *ResolvedQMgrName*is the name of the local queue manager hosting the transmission queue.

Creating dynamic queues

Use a dynamic queue when you do not need the queue after your application ends.

For example, you could use a dynamic queue for your reply-to queue. You specify the name of the reply-to queue in the *ReplyToQ* field of the MQMD structure when you put a message on a queue (see "Defining messages using the MQMD structure" on page 110).

To create a dynamic queue, you use a template known as a model queue, together with the MQOPEN call. You create a model queue using the WebSphere MQ commands or the operations and control panels. The dynamic queue that you create takes the attributes of the model queue.

When you call MQOPEN, specify the name of the model queue in the *ObjectName* field of the MQOD structure. When the call completes, the *ObjectName* field is set to the name of the dynamic queue that is created. Also, the *ObjectQMgrName* field is set to the name of the local queue manager.

You can specify the name of the dynamic queue that you create in three ways:

- Give the full name that you want in the *DynamicQName* field of the MQOD structure.
- Specify a prefix (fewer than 33 characters) for the name, and allow the queue manager to generate the rest of the name. This means that the queue manager generates a unique name, but you still have some control (for example, you might want each user to use a certain prefix, or you might want to give a special security classification to queues with a certain prefix in their name). To use this method, specify an asterisk (*) for the last non-blank character of the *DynamicQName* field. Do not specify a single asterisk (*) for the dynamic queue name.
- Allow the queue manager to generate the full name. To use this method, specify an asterisk (*) in the first character position of the *DynamicQName* field.

For more information about these methods, see the description of the *DynamicQName* field in the *WebSphere MQ Application Programming Reference*.

There is more information on dynamic queues in "Dynamic queues" on page 55.

Opening remote queues

A remote queue is a queue that is owned by a queue manager other than the one to which the application is connected.

To open a remote queue, use the MQOPEN call as for a local queue. You can specify the name of the queue as follows:

1. In the *ObjectName* field of the MQOD structure, specify the name of the remote queue as known to the *local* queue manager.

Note: Leave the *ObjectQMgrName* field blank in this case.

2. In the *ObjectName* field of the MQOD structure, specify the name of the remote queue, as known to the *remote* queue manager. In the *ObjectQMgrName* field, specify either:
   - The name of the transmission queue that has the same name as the remote queue manager. The name and case (uppercase, lowercase or a mixture) must match *exactly*.
   - The name of a queue manager alias object that resolves to the destination queue manager or the transmission queue.

   This tells the queue manager the destination of the message as well as the transmission queue that it needs to be put on to get there.
3. If *DefXmitQname* is supported, in the *ObjectName* field of the MQOD structure, specify the name of the remote queue as known by the *remote* queue manager.

   Note: Set the *ObjectQMgrName* field to the name of the remote queue manager (it cannot be left blank in this case).

Only local names are validated when you call MQOPEN; the last check is for the existence of the transmission queue to be used.

These methods are summarized in Table 5 on page 101.

Closing objects using the MQCLOSE call

To close an object, use the MQCLOSE call.

If the object is a queue, note the following:

- You do not need to empty a temporary dynamic queue before you close it.

  When you close a temporary dynamic queue, the queue is deleted, along with any messages that might still be on it. This is true even if there are uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue.
- On WebSphere MQ for z/OS, if you have any MQGET requests with an MQGMO_SET_SIGNAL option outstanding for that queue, they are canceled.
- If you opened the queue using the MQOO_BROWSE option, your browse cursor is destroyed.

Closure is unrelated to syncpoint, so you can close queues before or after syncpoint.

As input to the MQCLOSE call, you must supply:

- A connection handle. Use the same connection handle used to open it, or alternatively, for CICS applications on z/OS, you can specify the constant MQHC_DEF_HCONN (which has the value zero).
- The handle of the object that you want to close. Get this from the output of the MQOPEN call.
- MQCO_NONE in the *Options* field (unless you are closing a permanent dynamic queue).
- The control option to determine whether the queue manager should delete the queue even if there are still messages on it (when closing a permanent dynamic queue).

The output from MQCLOSE is:

- A completion code

- A reason code
- The object handle, reset to the value MQHO_UNUSABLE_HOBJ

Descriptions of the parameters of the MQCLOSE call are given in the *WebSphere MQ Application Programming Reference*.

Putting messages on a queue

Use the MQPUT call to put messages on the queue. You can use MQPUT repeatedly to put many messages on the same queue, following the initial MQOPEN call. Call MQCLOSE when you have finished putting all your messages on the queue.

If you want to put a single message on a queue and close the queue immediately afterwards, you can use the MQPUT1 call. MQPUT1 performs the same functions as the following sequence of calls:

- MQOPEN
- MQPUT
- MQCLOSE

Generally however, if you have more than one message to put on the queue, it is more efficient to use the MQPUT call. This depends on the size of the message and the platform that you are working on.

This chapter introduces putting messages to a queue, under these headings:

- "Putting messages on a local queue using the MQPUT call"
- "Putting messages on a remote queue" on page 115
- "Controlling context information" on page 115
- "Putting one message on a queue using the MQPUT1 call" on page 117
- "Distribution lists" on page 119
- "Some cases where the put calls fail" on page 124

Putting messages on a local queue using the MQPUT call

As input to the MQPUT call, you must supply:

- A connection handle (Hconn).
- A queue handle (Hobj).
- A description of the message that you want to put on the queue. This is in the form of a message descriptor structure (MQMD).
- Control information, in the form of a put-message options structure (MQPMO).
- The length of the data contained within the message (MQLONG).
- The message data itself.

The output from the MQPUT call is

- A reason code (MQLONG)
- A completion code (MQLONG)

If the call completes successfully, it also returns your options structure and your message descriptor structure. The call modifies your options structure to show the name of the queue and the queue manager to which the message was sent. If you

request that the queue manager generates a unique value for the identifier of the message you are putting (by specifying binary zero in the *MsgId* field of the MQMD structure), the call inserts the value in the *MsgId* field before returning this structure to you. Reset this value before you issue another MQPUT.

There is a description of the MQPUT call in the *WebSphere MQ Application Programming Reference*.

The following sections describe the information that you must supply as input to the MQPUT call.

Specifying handles

For the connection handle (*Hconn*) in CICS on z/OS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or you can use the connection handle returned by the MQCONN or MQCONNX call. For other applications, always use the connection handle returned by the MQCONN or MQCONNX call.

Whatever environment you are working in, use the same queue handle (*Hobj*) that is returned by the MQOPEN call.

Defining messages using the MQMD structure

The message descriptor structure (MQMD) is an input/output parameter for the MQPUT and MQPUT1 calls. Use it to define the message you are putting on a queue.

If MQPRI_PRIORITY_AS_Q_DEF or MQPER_PERSISTENCE_AS_Q_DEF is specified for the message and the queue is a cluster queue, the values used are those of the queue to which the MQPUT resolves. If that queue is disabled for MQPUT, the call will fail. See *WebSphere MQ Queue Manager Clusters* for more information.

Note: Use MQPMO_NEW_MSG_ID and MQPMO_NEW_CORREL_ID before putting a new message to ensure that the *MsgId* and *CorrelId* are unique. The values in these fields are returned on a successful MQPUT.

There is an introduction to the message properties that MQMD describes in "WebSphere MQ messages" on page 16, and there is a description of the structure itself in the *WebSphere MQ Application Programming Reference*.

Specifying options using the MQPMO structure

Use the MQPMO (Put Message Option) structure to pass options to the MQPUT and MQPUT1 calls.

The following sections give you help on filling in the fields of this structure. There is a description of the structure in the *WebSphere MQ Application Programming Reference*.

The fields of the structure include:

- *StrucId*
- *Version*
- *Options*
- *Context*
- *ResolvedQName*

- *ResolvedQMgrName*

These fields are described below.

StrucId
This identifies the structure as a put-message options structure. This is a 4-character field. Always specify MQPMO_STRUC_ID.

Version
This describes the version number of the structure. The default is MQPMO_VERSION_1. If you enter MQPMO_VERSION_2, you can use distribution lists (see "Distribution lists" on page 119). If you enter MQPMO_VERSION_3, you can use message handles and message properties. If you enter MQPMO_CURRENT_VERSION, your application is set always to use the most recent level.

Options
This controls the following:

- Whether the put operation is included in a unit of work
- How much context information is associated with a message
- Where the context information is taken from
- Whether the call fails if the queue manager is in a quiescing state
- Whether grouping or segmentation is allowed
- Generation of a new message identifier and correlation identifier
- The order in which messages and segments are put on a queue
- Whether to resolve local queue names

If you leave the *Options* field set to the default value (MQPMO_NONE), the message you put has default context information associated with it.

Also, the way that the call operates with syncpoints is determined by the platform. The syncpoint control default is yes in z/OS; for other platforms, it is no.

Context
This states the name of the queue handle that you want context information to be copied from (if requested in the *Options* field).

For an introduction to message context, see "Message context" on page 46. For information about using the MQPMO structure to control the context information in a message, see "Controlling context information" on page 115.

ResolvedQName
This contains the name (after resolution of any alias name) of the queue that was opened to receive the message. This is an output field.

ResolvedQMgrName
This contains the name (after resolution of any alias name) of the queue manager that owns the queue in *ResolvedQName*. This is an output field.

The MQPMO can also accommodate fields required for distribution lists (see "Distribution lists" on page 119). If you want to use this facility, Version 2 of the MQPMO structure is used. This includes the following fields:

RecsPresent
This field contains the number of queues in the distribution list; that is, the number of Put Message Records (MQPMR) and corresponding Response Records (MQRR) present.

The value that you enter can be the same as the number of Object Records provided at MQOPEN. However, if the value is less than the number of Object Records provided on the MQOPEN call, or if you provide no Put Message Records, the values of the queues that are not defined are taken from the default values provided by the message descriptor. Also, if the value is greater than the number of Object Records provided, the excess Put Message Records are ignored.

You are recommended to do one of the following:

- If you want to receive a report or reply from each destination, enter the same value as appears in the MQOR structure and use MQPMRs containing *MsgId* fields. Either initialize these *MsgId* fields to zeros or specify MQPMO_NEW_MSG_ID.

  When you have put the message to the queue, *MsgId* values that the queue manager has created become available in the MQPMRs; you can use these to identify which destination is associated with each report or reply.
- If you do not want to receive reports or replies, choose one of the following:
  1. If you want to identify destinations that fail immediately, you might still want to enter the same value in the *RecsPresent* field as appears in the MQOR structure and provide MQRRs to identify these destinations. Do not specify any MQPMRs.
  2. If you do not want to identify failed destinations, enter zero in the *RecsPresent* field and do not provide MQPMRs nor MQRRs.

Note: If you are using MQPUT1, the number of Response Record Pointers and Response Record Offsets must be zero.

For a full description of Put Message Records (MQPMR) and Response Records (MQRR), see the *WebSphere MQ Application Programming Reference*.

PutMsgRecFields
This indicates which fields are present in each Put Message Record (MQPMR). For a list of these fields, see "Using the MQPMR structure" on page 123.

PutMsgRecOffset **and** *PutMsgRecPtr*
Pointers (typically in C) and offsets (typically in COBOL) are used to address the Put Message Records (see "Using the MQPMR structure" on page 123 for an overview of the MQPMR structure).

Use the *PutMsgRecPtr* field to specify a pointer to the first Put Message Record, or the *PutMsgRecOffset* field to specify the offset of the first Put Message Record. This is the offset from the start of the MQPMO. Depending on the *PutMsgRecFields* field, enter a nonnull value for either *PutMsgRecOffset* or *PutMsgRecPtr*.

ResponseRecOffset and ResponseRecPtr
You also use pointers and offsets to address the Response Records (see "Using the MQRR structure" on page 122 for further information about Response Records).

Use the *ResponseRecPtr* field to specify a pointer to the first Response Record, or the *ResponseRecOffset* field to specify the offset of the first Response Record. This is the offset from the start of the MQPMO structure. Enter a nonnull value for either *ResponseRecOffset* or *ResponseRecPtr*.

Note: If you are using MQPUT1 to put messages to a distribution list, *ResponseRecPtr* must be null or zero and *ResponseRecOffset* must be zero.

Version 3 of the MQPMO structure additionally includes the following fields:

OriginalMsgHandle
The use you can make of this field depends on the value of the *Action* field. If you are putting a new message with associated message properties, set this field to the message handle you previously created and set properties on. If you are forwarding, replying to, or generating a report in response to a previously retrieved message, this field contains the message handle of that message.

NewMsgHandle
If you specify a *NewMsgHandle*, any properties associated with the handle override properties associated with the *OriginalMsgHandle*. For more information, see *WebSphere MQ Application Programming Reference*.

Action
Use this field to specify the type of put being performed. Possible values and their meanings are as follows:

MQACTP_NEW
This is a new message unrelated to any other.

MQACTP_FORWARD
This message was retrieved previously and is now being forwarded.

MQACTP_REPLY
This message is a reply to a previously retrieved message.

MQACTP_REPORT
This message is a report generated as a result of a previously retrieved message.

For more information, see *WebSphere MQ Application Programming Reference*.

PubLevel
If this message is a publication, you can set this field to determine which subscriptions receive it. Only subscriptions with a *SubLevel* less than or equal to this value will receive this publication. The default value is 9 which is the highest level and means that subscriptions with any *SubLevel* can receive this publication.

The data in your message

Give the address of the buffer that contains your data in the *Buffer* parameter of the MQPUT call. You can include anything in the data in your messages. The amount of data in the messages, however, affects the performance of the application that is processing them.

The maximum size of the data is determined by:

- The *MaxMsgLength* attribute of the queue manager
- The *MaxMsgLength* attribute of the queue on which you are putting the message
- The size of any message header added by WebSphere MQ (including the dead-letter header, MQDLH and the distribution list header, MQDH)

The *MaxMsgLength* attribute of the queue manager holds the size of message that the queue manager can process. This has a default of 100 MB for all WebSphere MQ products at V6 or higher.

To determine the value of this attribute, use the MQINQ call on the queue manager object. For large messages, you can change this value.

The *MaxMsgLength* attribute of a queue determines the maximum size of message that you can put on the queue. If you attempt to put a message with a size larger than the value of this attribute, your MQPUT call fails. If you are putting a message on a remote queue, the maximum size of message that you can successfully put is determined by the *MaxMsgLength* attribute of the remote queue, of any intermediate transmission queues that the message is put on along the route to its destination, and of the channels used.

For an MQPUT operation, the size of the message must be smaller than or equal to the *MaxMsgLength* attribute of both the queue and the queue manager. The values of these attributes are independent, but you are recommended to set the *MaxMsgLength* of the queue to a value less than or equal to that of the queue manager.

WebSphere MQ adds header information to messages in the following circumstances:

- When you put a message on a remote queue, WebSphere MQ adds a transmission header structure (MQXQH) to the message. This structure includes the name of the destination queue and its owning queue manager.
- If WebSphere MQ cannot deliver a message to a remote queue, it attempts to put the message on the dead-letter (undelivered-message) queue. It adds an MQDLH structure to the message. This structure includes the name of the destination queue and the reason that the message was put on the dead-letter queue.
- If you want to send a message to multiple destination queues, WebSphere MQ adds an MQDH header to the message. This describes the data that is present in a message, belonging to a distribution list, on a transmission queue. Consider this when choosing an optimum value for the maximum message length.
- If the message is a segment or a message in a group, WebSphere MQ might add an MQMDE.

These structures are described in the *WebSphere MQ Application Programming Reference*.

If your messages are of the maximum size allowed for these queues, the addition of these headers means that the put operations fail because the messages are now too big. To reduce the possibility of the put operations failing:

- Make the size of your messages smaller than the *MaxMsgLength* attribute of the transmission and dead-letter queues. Allow at least the value of the MQ_MSG_HEADER_LENGTH constant (more for large distribution lists).
- Make sure that the *MaxMsgLength* attribute of the dead-letter queue is set to the same as the *MaxMsgLength* of the queue manager that owns the dead-letter queue.

The attributes for the queue manager and the message queuing constants are described in the *WebSphere MQ Application Programming Reference*.

For information on how undelivered messages are handled in a distributed queuing environment, see *WebSphere MQ Intercommunications*.

Putting messages: Using message handles

Two message handles are available in the MQPMO structure, *OriginalMsgHandle* and *NewMsgHandle*. The relationship between these message handles is defined by the value of the MQPMO *Action* field.

For full details see the description of the Action field in the *WebSphere MQ Application Programming Reference*. A message handle is not necessarily required in order to put a message. Its purpose is to associate properties with a message, so it is required only if you are using message properties.

Putting messages on a remote queue

When you want to put a message on a remote queue (that is, a queue owned by a queue manager other than the one to which your application is connected) rather than a local queue, the only extra consideration is how you specify the name of the queue when you open it. This is described in "Opening remote queues" on page 107. There is no change to how you use the MQPUT or MQPUT1 call for a local queue.

For more information on using remote and transmission queues, see *WebSphere MQ Intercommunications*.

Setting properties of a message

Call MQSETMP for each property you want to set. When you put the message set the message handle and action fields of the MQPMO structure.

To associate properties with a message, the message must have a message handle. Create a message handle using the MQCRTMH function call. Call MQSETMP specifying this message handle for each property you want to set. A sample program, amqsstma.c, is provided to illustrate the use of MQSETMP.

If this is a new message, when you put it to a queue, using MQPUT or MQPUT1, set the OriginalMsgHandle field in the MQPMO to the value of this message handle, and set the MQPMO Action field to MQACTP_NEW (this is the default value).

If this is a message you have previously retrieved, and you are now forwarding or replying to it or sending a report in response to it, put the original message handle in the OriginalMsgHandle field of the MQPMO and the new message handle in the NewMsgHandle field. Set the Action field to MQACTP_FORWARD, MQACTP_REPLY, or MQACTP_REPORT, as appropriate.

For more information about the message handle and Action fields, see the *WebSphere MQ Application Programming Reference*.

If you have properties in an MQRFH2 header from a message you have previously retrieved, you can convert them to message handle properties using the MQBUFMH call.

Controlling context information

To control context information, use the `Options` field in the MQPMO structure.

If you do not, the queue manager overwrites context information that might already be in the message descriptor with the identity and context information that it has generated for your message. This is the same as specifying the MQPMO_DEFAULT_CONTEXT option. You might want this default context information when you create a new message (for example, when processing user input from an inquiry screen).

If you want no context information associated with your message, use the MQPMO_NO_CONTEXT option.

The following topics explain the use of identity context, user context and all context.

Passing identity context

In general, programs should pass identity context information from message to message around an application until the data reaches its final destination.

Programs should change the origin context information each time that they change the data. However, applications that want to change or set any context information must have the appropriate level of authority. The queue manager checks this authority when the applications open the queues; they must have authority to use the appropriate context options for the MQOPEN call.

If your application gets a message, processes the data from the message, then puts the changed data into another message (possibly for processing by another application), the application must pass the identity context information from the original message to the new message. You can allow the queue manager to create the origin context information.

To save the context information from the original message, use the MQOO_SAVE_ALL_CONTEXT option when you open the queue for getting the message. This is in addition to any other options you use with the MQOPEN call. Note, however, that you cannot save context information if you only browse the message.

When you create the second message:

- Open the queue using the MQOO_PASS_IDENTITY_CONTEXT option (in addition to the MQOO_OUTPUT option).
- In the *Context* field of the put-message options structure, give the handle of the queue from which you saved the context information.
- In the *Options* field of the put-message options structure, specify the MQPMO_PASS_IDENTITY_CONTEXT option.

Passing user context

You cannot choose to pass only user context. To pass user context when putting a message, specify MQPMO_PASS_ALL_CONTEXT. Any properties in the user context are passed in the same way as the origin context.

When an MQPUT or MQPUT1 takes place and the context is being passed, all properties in the user context are passed from the retrieved message to the put message. Any user context properties that the putting application has altered are put with their original values. Any user context properties that the putting application has deleted are restored in the put message. Any user context properties that the putting application has added to the message are retained.

Passing all context

If your application gets a message, and puts the message data (unchanged) into another message, the application must pass all (identity, origin, and user) context information from the original message to the new message. An example of an application that might do this is a message mover, which moves messages from one queue to another.

Follow the same procedure as for passing identity context, except that you use the MQOPEN option MQOO_PASS_ALL_CONTEXT and the put-message option MQPMO_PASS_ALL_CONTEXT.

Setting identity context

If you want to set the identity context information for a message:

- Open the queue using the MQOO_SET_IDENTITY_CONTEXT option.
- Put the message on the queue, specifying the MQPMO_SET_IDENTITY_CONTEXT option. In the message descriptor, specify whatever identity context information you require.

Note: When you set some (but not all) of the identity context fields using the MQOO_SET_IDENTITY_CONTEXT and MQPMO_SET_IDENTITY_CONTEXT options, it is important to realize that the queue manager *does not* set any of the other fields.

Setting user context

To set a property in the user context, set the Context field of the message property descriptor (MQPD) to MQPD_USER_CONTEXT when you make the MQSETMP call..

You do not need any special authority to set a property in the user context. User context has no MQOO_SET_* or MQPMO_SET_* context options

Setting all context

If you want to set both the identity and the origin context information for a message:

1. Open the queue using the MQOO_SET_ALL_CONTEXT option.
2. Put the message on the queue, specifying the MQPMO_SET_ALL_CONTEXT option. In the message descriptor, specify whatever identity and origin context information you require.

Appropriate authority is needed for each type of context setting.

Putting one message on a queue using the MQPUT1 call

Use the MQPUT1 call when you want to close the queue immediately after you have put a single message on it. For example, a server application is likely to use the MQPUT1 call when it is sending a reply to each of the different queues.

MQPUT1 is functionally equivalent to calling MQOPEN followed by MQPUT, followed by MQCLOSE. The only difference in the syntax for the MQPUT and MQPUT1 calls is that for MQPUT you specify an object handle, whereas for MQPUT1 you specify an object descriptor structure (MQOD) as defined in MQOPEN (see “Identifying objects (the MQOD structure)” on page 101). This is because you need to give information to the MQPUT1 call about the queue that it has to open, whereas when you call MQPUT, the queue must already be open.

As input to the MQPUT1 call, you must supply:

- A connection handle.
- A description of the object that you want to open. This is in the form of an object descriptor structure (MQOD).
- A description of the message that you want to put on the queue. This is in the form of a message descriptor structure (MQMD).
- Control information in the form of a put-message options structure (MQPMO).
- The length of the data contained within the message (MQLONG).
- The address of the message data.

The output from MQPUT1 is:

- A completion code
- A reason code

If the call completes successfully, it also returns your options structure and your message descriptor structure. The call modifies your options structure to show the name of the queue and the queue manager to which the message was sent. If you request that the queue manager generate a unique value for the identifier of the message that you are putting (by specifying binary zero in the *MsgId* field of the MQMD structure), the call inserts the value in the *MsgId* field before returning this structure to you.

Note: You cannot use MQPUT1 with a model queue name; however, once a model queue has been opened, you can issue an MQPUT1 to the dynamic queue.

The six input parameters for MQPUT1 are:

Hconn This is a connection handle. For CICS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNX call. For other programs, always use the connection handle returned by the MQCONN or MQCONNX call.

ObjDesc
This is an object descriptor structure (MQOD).

In the *ObjectName* and *ObjectQMgrName* fields, give the name of the queue on which you want to put a message, and the name of the queue manager that owns this queue.

The *DynamicQName* field is ignored for the MQPUT1 call because it cannot use model queues.

Use the *AlternateUserId* field if you want to nominate an alternate user identifier that is to be used to test authority to open the queue.

MsgDesc
This is a message descriptor structure (MQMD). As with the MQPUT call, use this structure to define the message that you are putting on the queue.

PutMsgOpts
This is a put-message options structure (MQPMO). Use it as you would for the MQPUT call (see "Specifying options using the MQPMO structure" on page 110).

When the *Options* field is set to zero, the queue manager uses your own user ID when it performs tests for authority to access the queue. Also, the

queue manager ignores any alternate user identifier given in the *AlternateUserId* field of the MQOD structure.

BufferLength
This is the length of your message.

Buffer This is the buffer area that contains the text of your message.

When you use clusters, MQPUT1 operates as though MQOO_BIND_NOT_FIXED is in effect. Applications must use the resolved fields in the MQPMO structure rather than the MQOD structure to determine where the message was sent. See *WebSphere MQ Queue Manager Clusters* for more information.

There is a description of the MQPUT1 call in the *WebSphere MQ Application Programming Reference*.

Distribution lists

Not supported on WebSphere MQ for z/OS.

Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. Multiple queues can be opened using a single MQOPEN and a message can then be put to each of those queues using a single MQPUT. Some generic information from the MQI structures used for this process can be superseded by specific information relating to the individual destinations included in the distribution list.

When an MQOPEN call is issued, generic information is taken from the Object Descriptor (MQOD). If you specify MQOD_VERSION_2 in the *Version* field and a value greater than zero in the *RecsPresent* field, the *Hobj* can be defined as a handle of a list (of one or more queues) rather than of a queue. In this case, specific information is given through the object records (MQORs), which give details of destination (that is, *ObjectName* and *ObjectQMgrName*).

The object handle (*Hobj*) is passed to the MQPUT call, allowing you to put to a list rather than to a single queue.

When a message is put on the queues (MQPUT), generic information is taken from the Put Message Option structure (MQPMO) and the Message Descriptor (MQMD). Specific information is given in the form of Put Message Records (MQPMRs).

Response Records (MQRR) can receive a completion code and reason code specific to each destination queue.

Figure 9 on page 120 shows how distribution lists work.

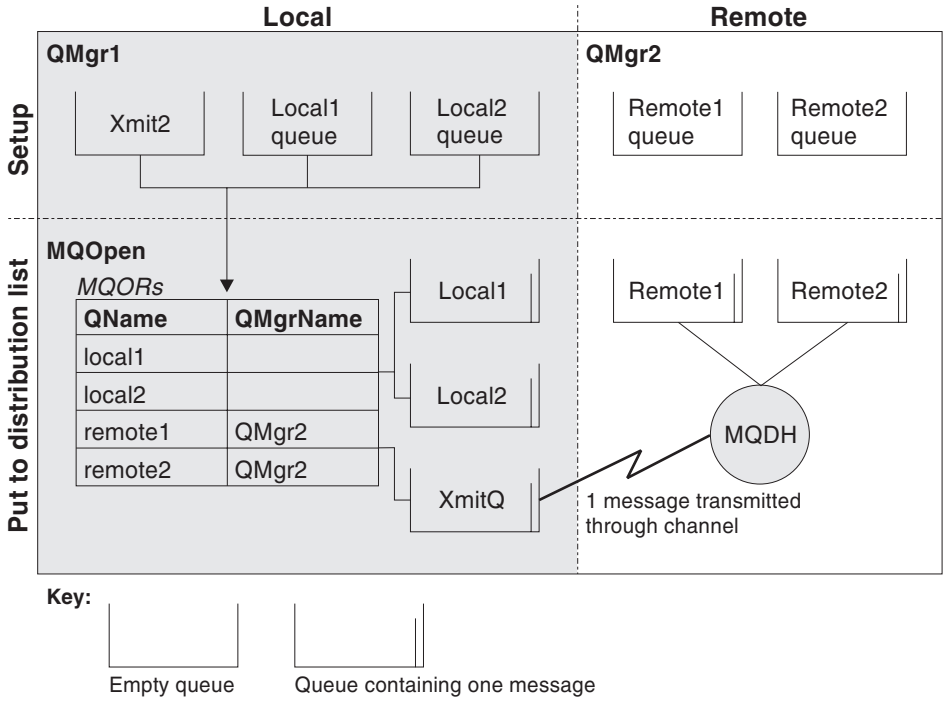


Figure 9. How distribution lists work. This diagram shows that *one* message is transmitted through the channel and can be put on more than one remote queue.

Opening distribution lists

Use the MQOPEN call to open a distribution list, and use the options of the call to specify what you want to do with the list.

As input to MQOPEN, you must supply:

- A connection handle (see "Putting messages on a queue" on page 109 for a description)
- Generic information in the Object Descriptor structure (MQOD)
- The name of each queue that you want to open, using the Object Record structure (MQOR)

The output from MQOPEN is:

- An object handle that represents your access to the distribution list
- A generic completion code
- A generic reason code
- Response Records (optional), containing a completion code and reason for each destination

Using the MQOD structure:

Use the MQOD structure to identify the queues that you want to open.

To define a distribution list, you must specify MQOD_VERSION_2 in the *Version* field, a value greater than zero in the *RecsPresent* field, and MQOT_Q in the *ObjectType* field. See MQOD - Object descriptor for a description of all the fields of the MQOD structure.

Using the MQOR structure:

Provide an MQOR structure for each destination.

The structure contains the destination queue and queue manager names. The *ObjectName* and *ObjectQMgrName* fields in the MQOD are not used for distribution lists. There must be one or more object records. If the *ObjectQMgrName* is left blank, the local queue manager is used. See the *WebSphere MQ Application Programming Reference* for further information about these fields.

You can specify the destination queues in two ways:

- By using the offset field *ObjectRecOffset*.

  In this case, the application must declare its own structure containing an MQOD structure, followed by the array of MQOR records (with as many array elements as are needed), and set *ObjectRecOffset* to the offset of the first element in the array from the start of the MQOD. Ensure that this offset is correct.

  Use of built-in facilities provided by the programming language is recommended, if these are available in all the environments in which the application runs. The following illustrates this technique for the COBOL programming language:

```
 01  MY-OPEN-DATA.
     02 MY-MQOD.
        COPY CMQODV.
     02 MY-MQOR-TABLE OCCURS 100 TIMES.
        COPY CMQORV.
     MOVE LENGTH OF MY-MQOD TO MQOD-OBJECTRECOFFSET.
```

  Alternatively, use the constant MQOD_CURRENT_LENGTH if the programming language does not support the necessary built-in facilities in all the environments concerned. The following illustrates this technique:

```
 01  MY-MQ-CONSTANTS.
     COPY CMQV.
 01  MY-OPEN-DATA.
     02 MY-MQOD.
        COPY CMQODV.
     02 MY-MQOR-TABLE OCCURS 100 TIMES.
        COPY CMQORV.
     MOVE MQOD-CURRENT-LENGTH TO MQOD-OBJECTRECOFFSET.
```

  However, this works correctly only if the MQOD structure and the array of MQOR records are contiguous; if the compiler inserts skip bytes between the MQOD and the MQOR array, these must be added to the value stored in *ObjectRecOffset*.

  Using *ObjectRecOffset* is recommended for programming languages that do not support the pointer data type, or that implement the pointer data type in a way that is not portable to different environments (for example, the COBOL programming language).

- By using the pointer field *ObjectRecPtr*.

  In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ObjectRecPtr* to the address of the array. The following illustrates this technique for the C programming language:

```
MQOD MyMqod;
MQOR MyMqor[100];
MyMqod.ObjectRecPtr = MyMqor;
```

  Using *ObjectRecPtr* is recommended for programming languages that support the pointer data type in a way that is portable to different environments (for example, the C programming language).

Whichever technique you choose, you must use one of *ObjectRecOffset* and *ObjectRecPtr*; the call fails with reason code MQRC_OBJECT_RECORDS_ERROR if both are zero, or both are nonzero.

Using the MQRR structure:

These structures are destination specific; each Response Record contains a *CompCode* and *Reason* field for each queue of a distribution list. You must use this structure to enable you to distinguish where any problems lie.

For example, if you receive a reason code of MQRC_MULTIPLE_REASONS and your distribution list contains five destination queues, you will not know which queues the problems apply to if you do not use this structure. However, if you have a completion code and reason code for each destination, you can locate the errors more easily.

See the *WebSphere MQ Application Programming Reference* for further information about the MQRR structure.

Figure 10 shows how you can open a distribution list in C.

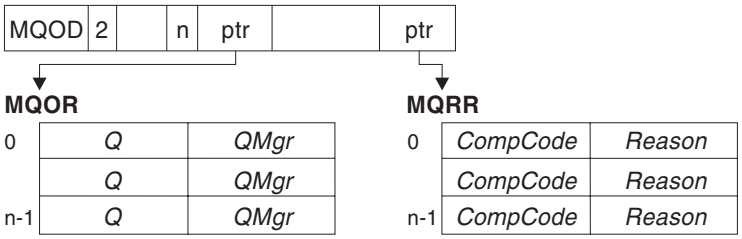


Figure 10. Opening a distribution list in C. The MQOD uses pointers to the MQOR and MQRR structures.

Figure 11 shows how you can open a distribution list in COBOL.

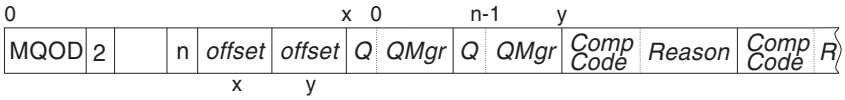


Figure 11. Opening a distribution list in COBOL. The MQOD uses offsets in COBOL.

Using the MQOPEN options:

You can specify the following options when opening a distribution list:

- MQOO_OUTPUT
- MQOO_FAIL_IF_QUIESCING (optional)
- MQOO_ALTERNATE_USER_AUTHORITY (optional)
- MQOO_*_CONTEXT (optional)

See "Opening and closing objects" on page 99 for a description of these options.

Putting messages to a distribution list

To put messages to a distribution list, you can use MQPUT or MQPUT1.

As input, you must supply:

- A connection handle (see "Putting messages on a queue" on page 109 for a description).
- An object handle. If a distribution list is opened using MQOPEN, the *Hobj* allows you only to put to the list.
- A message descriptor structure (MQMD). See the *WebSphere MQ Application Programming Reference* for a description of this structure.
- Control information in the form of a put-message option structure (MQPMO). See "Specifying options using the MQPMO structure" on page 110 for information about filling in the fields of the MQPMO structure.
- Control information in the form of Put Message Records (MQPMR).
- The length of the data contained within the message (MQLONG).
- The message data itself.

The output is:

- A completion code
- A reason code
- Response Records (optional)

Using the MQPMR structure:

This structure is optional and gives destination-specific information for some fields that you might want to identify differently from those already identified in the MQMD.

For a description of these fields, see the *WebSphere MQ Application Programming Reference*.

The content of each record depends on the information given in the *PutMsgRecFields* field of the MQPMO. For example, in the sample program AMQSPTL0.C (see "The Distribution List sample program" on page 405 for a description) showing the use of distribution lists, the sample chooses to provide values for *MsgId* and *CorrelId* in the MQPMR. This section of the sample program looks like this:

```
typedef struct
{
MQBYTE24 MsgId;
MQBYTE24 CorrelId;
} PutMsgRec;...
/**********************
MQLONG PutMsgRecFields=MQPMRF_MSG_ID | MQPMRF_CORREL_ID;
```

This implies that *MsgId* and *CorrelId* are provided for each destination of a distribution list. The Put Message Records are provided as an array.

Figure 12 on page 124 shows how you can put a message to a distribution list in C.

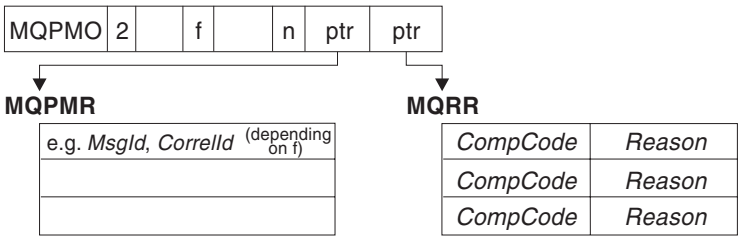


Figure 12. Putting a message to a distribution list in C. The MQPMO uses pointers to the MQPMR and MQRR structures.

Figure 13 shows how you can put a message to a distribution list in COBOL.

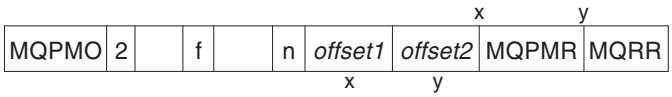


Figure 13. Putting a message to a distribution list in COBOL. The MQPMO uses offsets in COBOL.

Using MQPUT1:

If you are using MQPUT1, consider the following:

1. The values of the *ResponseRecOffset* and *ResponseRecPtr* fields must be null or zero.
2. The Response Records, if required, must be addressed from the MQOD.

Some cases where the put calls fail

If certain attributes of a queue are changed using the FORCE option on a command during the interval between you issuing an MQOPEN and an MQPUT call, the MQPUT call fails and returns the MQRC_OBJECT_CHANGED reason code.

The queue manager marks the object handle as being no longer valid. This also happens if the changes are made while an MQPUT1 call is being processed, or if the changes apply to any queue to which the queue name resolves. The attributes that affect the handle in this way are listed in the description of the MQOPEN call in the *WebSphere MQ Application Programming Reference*. If your call returns the MQRC_OBJECT_CHANGED reason code, close the queue, reopen it, then try to put a message again.

If put operations are inhibited for a queue on which you are attempting to put messages (or any queue to which the queue name resolves), the MQPUT or MQPUT1 call fails and returns the MQRC_PUT_INHIBITED reason code. You might be able to put a message successfully if you attempt the call at a later time, if the design of the application is such that other programs change the attributes of queues regularly.

Furthemore, if the queue that you are trying to put your message on is full, the MQPUT or MQPUT1 call fails and returns MQRC_Q_FULL.

If a dynamic queue (either temporary or permanent) has been deleted, MQPUT calls using a previously-acquired object handle fail and return the MQRC_Q_DELETED reason code. In this situation, it is good practice to close the object handle as it is no longer of any use to you.

In the case of distribution lists, multiple completion codes and reason codes can occur in a single request. These cannot be handled using only the *CompCode* and *Reason* output fields on MQOPEN and MQPUT.

When you use distribution lists to put messages to multiple destinations, the Response Records contain the specific *CompCode* and *Reason* for each destination. If you receive a completion code of MQCC_FAILED, no message is put on any destination queue successfully. If the completion code is MQCC_WARNING, the message is successfully put on one or more of the destination queues. If you receive a return code of MQRC_MULTIPLE_REASONS, the reason codes are not all the same for every destination. Therefore, it is recommended to use the MQRR structure so that you can determine which queue or queues caused an error and the reasons for each.

Getting messages from a queue

You can get messages from a queue in two ways:

1. You can *remove* a message from the queue so that other programs can no longer see it.
2. You can *copy* a message, leaving the original message on the queue. This is known as *browsing*. You can remove the message once you have browsed it.

In both cases, you use the MQGET call, but first your application must be connected to the queue manager, and you must use the MQOPEN call to open the queue (for input, browse, or both). These operations are described in "Connecting to and disconnecting from a queue manager" on page 91 and "Opening and closing objects" on page 99.

When you have opened the queue, you can use the MQGET call repeatedly to browse or remove messages on the same queue. Call MQCLOSE when you have finished getting all the messages that you want from the queue.

This chapter introduces getting messages from a queue, under these headings:

Getting messages from a queue using the MQGET call

The MQGET call gets a message from an open local queue. It cannot get a message from a queue on another system.

As input to the MQGET call, you must supply:

- A connection handle.

- A queue handle.
- A description of the message that you want to get from the queue. This is in the form of a message descriptor (MQMD) structure.
- Control information in the form of a Get Message Options (MQGMO) structure.
- The size of the buffer that you have assigned to hold the message (MQLONG).
- The address of the storage in which to put the message.

The output from MQGET is:

- A reason code
- A completion code
- The message in the buffer area that you specified, if the call completes successfully
- Your options structure, modified to show the name of the queue from which the message was retrieved
- Your message descriptor structure, with the contents of the fields modified to describe the message that was retrieved
- The length of the message (MQLONG)

There is a description of the MQGET call in the *WebSphere MQ Application Programming Reference*.

The following sections describe the information you must supply as input to the MQGET call.

Specifying connection handles

For CICS on z/OS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNX call. For other applications, always use the connection handle returned by the MQCONN or MQCONNX call.

Use the queue handle (*Hobj*) that is returned when you call MQOPEN.

Describing messages using the MQMD structure and the MQGET call

To identify the message that you want to get from a queue, use the message descriptor structure (MQMD).

This is an input/output parameter for the MQGET call. There is an introduction to the message properties that MQMD describes in "WebSphere MQ messages" on page 16, and there is a description of the structure itself in the *WebSphere MQ Application Programming Reference*.

If you know which message you want to get from the queue, see "Getting a particular message" on page 138.

If you do not specify a particular message, MQGET retrieves the *first* message in the queue. "The order in which messages are retrieved from a queue" on page 131 describes how the priority of a message, the *MsgDeliverySequence* attribute of the queue, and the MQGMO_LOGICAL_ORDER option determine the order of the messages in the queue.

Note: If you want to use MQGET more than once (for example, to step through the messages in the queue), you must set the *MsgId* and *CorrelId* fields of this structure to null after each call. This clears these fields of the identifiers of the message that was retrieved.

However, if you want to group your messages, the *GroupId* must be the same for messages in the same group, so that the call looks for a message having the same identifiers as the previous message in order to make up the whole group.

Specifying MQGET options using the MQGMO structure

The MQGMO structure is an input/output variable for passing options to the MQGET call. The following sections help you to complete some of the fields of this structure.

There is a description of the MQGMO structure in the *WebSphere MQ Application Programming Reference*.

StrucId

StrucId is a 4-character field used to identify the structure as a get-message options structure. Always specify MQGMO_STRUC_ID.

Version

Version describes the version number of the structure. MQGMO_VERSION_1 is the default. If you want to use the Version 2 fields or retrieve messages in logical order, specify MQGMO_VERSION_2. If you want to use the Version 3 fields or retrieve messages in logical order, specify MQGMO_VERSION_3. MQGMO_CURRENT_VERSION sets your application to use the most recent level.

Options

Within your code, you can select the options in any order; each option is represented by a bit in the *Options* field.

The *Options* field controls:

- Whether the MQGET call waits for a message to arrive on the queue before it completes (see "Waiting for messages" on page 149)
- Whether the get operation is included in a unit of work.
- Whether a nonpersistent message is retrieved outside syncpoint, allowing fast messaging
- On WebSphere MQ for z/OS, whether the message retrieved is marked as skipping backout (see "Skipping backout" on page 152)
- Whether the message is removed from the queue, or merely browsed
- Whether to select a message by using a browse cursor or by other selection criteria
- Whether the call succeeds even if the message is longer than your buffer
- On WebSphere MQ for z/OS, whether to allow the call to complete. This option also sets a signal to indicate that you want to be notified when a message arrives
- Whether the call fails if the queue manager is in a quiescing state
- On WebSphere MQ for z/OS, whether the call fails if the connection is in a quiescing state
- Whether application message data conversion is required (see "Application data conversion" on page 154)
- The order in which messages and (with the exception of WebSphere MQ for z/OS) segments are retrieved from a queue

- Except on WebSphere MQ for z/OS, whether complete, logical messages only are retrievable
- Whether messages in a group can be retrieved only when *all* messages in the group are available
- Except on WebSphere MQ for z/OS, whether segments in a logical message can be retrieved only when *all* segments in the logical message are available

If you leave the *Options* field set to the default value (MQGMO_NO_WAIT), the MQGET call operates this way:

- If there is no message matching your selection criteria on the queue, the call does not wait for a message to arrive, but completes immediately. Also, in WebSphere MQ for z/OS, the call does not set a signal requesting notification when such a message arrives.
- The way that the call operates with syncpoints is determined by the platform:

Platform	Under syncpoint control
i5/OS	No
UNIX systems	No
z/OS	Yes
Windows systems	No

- On WebSphere MQ for z/OS, the message retrieved is not marked as skipping backout.
- The selected message is removed from the queue (not browsed).
- No application message data conversion is required.
- The call fails if the message is longer than your buffer.

WaitInterval

The *WaitInterval* field specifies the maximum time (in milliseconds) that the MQGET call waits for a message to arrive on the queue when you use the MQGMO_WAIT option. If no message arrives within the time specified in *WaitInterval*, the call completes and returns a reason code showing that there was no message that matched your selection criteria on the queue.

On WebSphere MQ for z/OS, if you use the MQGMO_SET_SIGNAL option, the *WaitInterval* field specifies the time for which the signal is set.

For more information on these options, see "Waiting for messages" on page 149 and "Signaling" on page 150.

Signal1

Signal1 is supported on WebSphere MQ for z/OS and MQSeries® for Compaq NonStop Kernel only.

If you use the MQGMO_SET_SIGNAL option to request that your application is notified when a suitable message arrives, you specify the type of signal in the *Signal1* field. In WebSphere MQ on all other platforms, the *Signal1* field is reserved and its value is not significant.

For more information, see "Signaling" on page 150.

Signal2

The *Signal2* field is reserved on all platforms and its value is not significant.

For more information, see “Signaling” on page 150.

ResolvedQName
ResolvedQName is an output field in which the queue manager returns the name of the queue (after resolution of any alias) from which the message was retrieved.

MatchOptions
MatchOptions controls the selection criteria for MQGET.

GroupStatus
GroupStatus indicates whether the message that you have retrieved is in a group.

SegmentStatus
SegmentStatus indicates whether the item that you have retrieved is a segment of a logical message.

Segmentation
Segmentation indicates whether segmentation is allowed for the message retrieved.

MsgToken

MsgToken uniquely identifies a message.

For more information, see “WebSphere MQ Workflow” on page 281.

ReturnedLength
ReturnedLength is an output field in which the queue manager returns the length of message data returned (in bytes).

MsgHandle
The handle to a message that is to be populated with the properties of the message being retrieved from the queue. The handle has previously been created by an MQCRTMH call. Any properties already associated with the handle are cleared before retrieving a message.

Retrieving properties of a message

You can either retrieve message properties using a message handle or MQRFH2 headers. Use the MQGMO Options field to indicate how you want properties to be returned. Use MQINQMP to return properties associated with a message handle to your application.

If you set MQGMO_PROPERTIES_IN_HANDLE and set the MsgHandle in your MQGET call to a valid value, the message properties are made available using that message handle. If you set MQGMO_PROPERTIES_FORCE_MQRFH2, the message properties are returned in MQRFH2 headers. If you set MQGMO_PROPERTIES_AS_Q_DEF, properties are represented as defined by the PropertyControl queue attribute; however, if a MsgHandle is provided this option is ignored and the properties of the message are made available on the MsgHandle, unless the value of the PropertyControl queue attribute is MQPROP_FORCE_MQRFH2.

When you have associated the message properties with the message handle, you can return each property to your application using MQINQMP. A sample program, amqsiqma.c, is provided to illustrate the use of MQINQMP.

You should normally return properties using a message handle. If you already use properties in MQRFH2 headers in applications using earlier versions of WebSphere MQ, you can continue to do so by using

MQGMO_PROPERTIES_FORCE_MQRFH2 or MQPROP_FORCE_MQRFH2. You can also use the MQMHBUF call to convert properties from a message handle to MQRFH2 format.

If you set properties using a message handle, an application connected to an earlier version of Websphere MQ can retrieve them using MQRFH2 headers.

Websphere MQ Version 7.0 clients connected to queue managers at an earlier version can retrieve properties using message handles, although those properties were set using MQRFH2 headers.

Specifying the size of the buffer area

In the *BufferLength* parameter of the MQGET call, specify the size of the buffer area to hold the message data that you retrieve. You decide how big this should be in three ways:

1. You might already know what length of messages to expect from this program. If so, specify a buffer of this size.

   However, you can use the MQGMO_ACCEPT_TRUNCATED_MSG option in the MQGMO structure if you want the MQGET call to complete even if the message is too big for the buffer. In this case:

   - The buffer is filled with as much of the message as it can hold
   - The call returns a warning completion code
   - The message is removed from the queue (discarding the remainder of the message), or the browse cursor is advanced (if you are browsing the queue)
   - The real length of the message is returned in *DataLength*

   Without this option, the call still completes with a warning, but it does not remove the message from the queue (or advance the browse cursor).
2. Estimate a size for the buffer (or even specify a size of zero bytes) and *do not* use the MQGMO_ACCEPT_TRUNCATED_MSG option. If the MQGET call fails (for example, because the buffer is too small), the length of the message is returned in the *DataLength* parameter of the call. (The buffer is still filled with as much of the message as it can hold, but the processing of the call is not completed.) Store the *MsgId* of this message, then repeat the MQGET call, specifying a buffer area of the correct size, and the *MsgId* that you noted from the first call.

   If your program is serving a queue that is also being served by other programs, one of those other programs might remove the message that you want before your program can issue another MQGET call. Your program could waste time searching for a message that no longer exists. To avoid this, first browse the queue until you find the message that you want, specifying a *BufferLength* of zero and using the MQGMO_ACCEPT_TRUNCATED_MSG option. This positions the browse cursor under the message that you want. You can then retrieve the message by calling MQGET again, specifying the MQGMO_MSG_UNDER_CURSOR option. If another program removes the message between your browse and removal calls, your second MQGET fails immediately (without searching the whole queue), because there is no message under your browse cursor.
3. The *MaxMsgLength queue* attribute determines the maximum length of messages accepted for that queue; the *MaxMsgLength queue manager* attribute determines the maximum length of messages accepted for that queue manager. If you do

not know what length of message to expect, you can inquire about the *MaxMsgLength* attribute (using the MQINQ call), then specify a buffer of this size.

Try to make the buffer size as close as possible to the actual message size to avoid reduced performance.

For further information about the *MaxMsgLength* attribute, see "Increasing the maximum message length" on page 143.

The order in which messages are retrieved from a queue

You can control the order in which you retrieve messages from a queue. This section looks at the options.

Priority

A program can assign a priority to a message when it puts the message on a queue (see "Message priorities" on page 26). Messages of equal priority are stored in a queue in order of arrival, not the order in which they are committed.

The queue manager maintains queues either in strict FIFO (first in, first out) sequence, or in FIFO within priority sequence. This depends on the setting of the *MsgDeliverySequence* attribute of the queue. When a message arrives on a queue, it is inserted immediately following the last message that has the same priority.

Programs can either get the first message from a queue, or they can get a particular message from a queue, ignoring the priority of those messages. For example, a program might want to process the reply to a particular message that it sent earlier. For more information, see "Getting a particular message" on page 138.

If an application puts a sequence of messages on a queue, another application can retrieve those messages in the same order that they were put, provided:

- The messages all have the same priority
- The messages were all put within the same unit of work, or all put outside a unit of work
- The queue is local to the putting application

If these conditions are not met, and the applications depend on the messages being retrieved in a certain order, the applications must either include sequencing information in the message data, or establish a means of acknowledging receipt of a message before the next one is sent.

On WebSphere MQ for z/OS, you can use the queue attribute, *IndexType*, to increase the speed of MQGET operations on the queue. For more information, see "Type of index" on page 142.

Logical and physical ordering

Messages on queues can occur (within each priority level) in *physical* or *logical* order.

Physical order is the order in which messages arrive on a queue. Logical order is when all of the messages and segments within a group are in their logical sequence, adjacent to each other, in the position determined by the physical position of the first item belonging to the group.

For a description of groups, messages, and segments, see "Message groups" on page 43. These physical and logical orders can differ because:

- Groups can arrive at a destination at similar times from different applications, therefore losing any distinct physical order.
- Even within a single group, messages can get out of order because of rerouting or delay of some of the messages in the group.

For example, the logical order might look like Figure Figure 14:

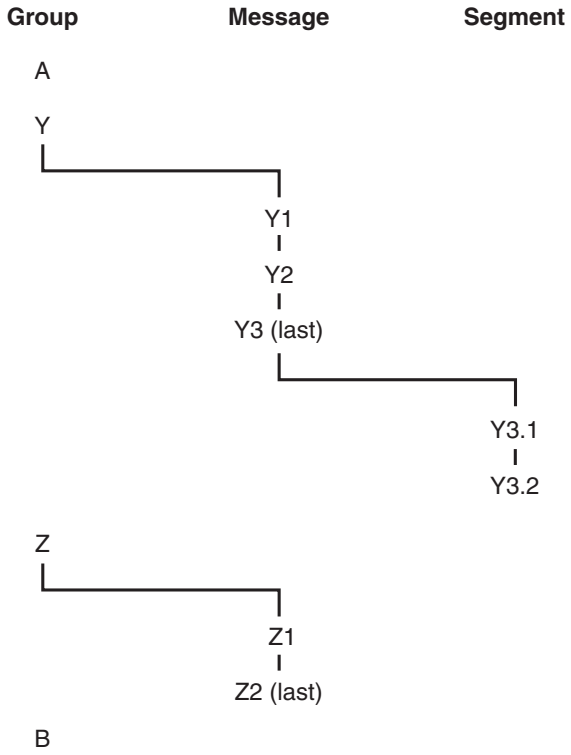


Figure 14. Logical order on a queue

These messages would appear in the following logical order on a queue:

1. Message A (not in a group)
2. Logical message 1 of group Y
3. Logical message 2 of group Y
4. Segment 1 of (last) logical message 3 of group Y
5. (Last) segment 2 of (last) logical message 3 of group Y
6. Logical message 1 of group Z
7. (Last) logical message 2 of group Z
8. Message B (not in a group)

The physical order, however, might be entirely different. As stated in topic "Logical and physical ordering" on page 131, the physical position of the *first* item within each group determines the logical position of the whole group. For example, if groups Y and Z arrived at similar times, and message 2 of group Z overtook message 1 of the same group, the physical order would look like Figure Figure 15 on page 133:

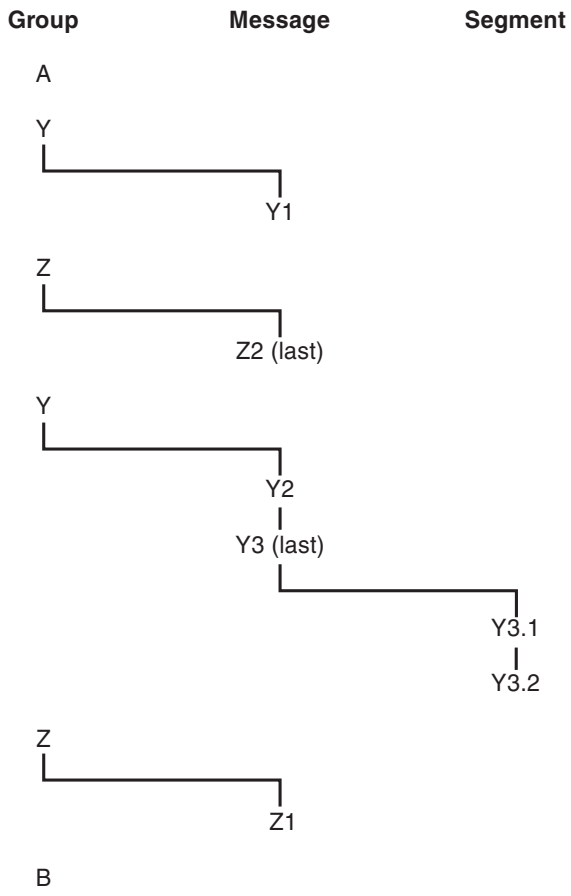


Figure 15. Physical order on a queue

These messages appear in the following physical order on the queue:

1. Message A (not in a group)
2. Logical message 1 of group Y
3. Logical message 2 of group Z
4. Logical message 2 of group Y
5. Segment 1 of (last) logical message 3 of group Y
6. (Last) segment 2 of (last) logical message 3 of group Y
7. Logical message 1 of group Z
8. Message B (not in a group)

Note: On WebSphere MQ for z/OS, the physical order of messages on the queue is not guaranteed if the queue is indexed by GROUPID.

When getting messages, you can specify MQGMO_LOGICAL_ORDER to retrieve messages in logical rather than physical order.

If you issue an MQGET call with MQGMO_BROWSE_FIRST and MQGMO_LOGICAL_ORDER, subsequent MQGET calls with MQGMO_BROWSE_NEXT must also specify this option. Conversely, if the MQGET with MQGMO_BROWSE_FIRST does not specify MQGMO_LOGICAL_ORDER, neither must the following MQGETs with MQGMO_BROWSE_NEXT.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue is separate from the group and segment information that the queue manager retains for MQGET calls that remove messages from the queue. When you specify MQGMO_BROWSE_FIRST, the queue manager ignores the group and segment information for browsing, and scans the queue as though there were no current group and no current logical message.

Note: Take special care if you use an MQGET call to browse *beyond the end* of a message group (or logical message not in a group) without specifying MQGMO_LOGICAL_ORDER. For example, if the last message in the group *precedes* the first message in the group on the queue, using MQGMO_BROWSE_NEXT to browse beyond the end of the group, specifying MQMO_MATCH_MSG_SEQ_NUMBER with *MsgSeqNumber* set to 1 (to find the first message of the next group) returns again the first message in the group already browsed. This could happen immediately, or a number of MQGET calls later (if there are intervening groups).

Avoid the possibility of an infinite loop by opening the queue *twice* for browse:

- Use the first handle to browse only the first message in each group.
- Use the second handle to browse only the messages within a specific group.
- Use the MQMO_* options to move the second browse cursor to the position of the first browse cursor, before browsing the messages in the group.
- Do not use the MQGMO_BROWSE_NEXT browse beyond the end of a group.

For further information about this, see the *WebSphere MQ Application Programming Reference*.

For most applications you will probably choose either logical or physical ordering when browsing. However, if you want to switch between these modes, remember that when you first issue a browse with MQGMO_LOGICAL_ORDER, your position within the logical sequence is established.

If the first item within the group is not present at this time, the group that you are in is not considered to be part of the logical sequence.

Once the browse cursor is within a group, it can continue within the same group, even if the first message is removed. Initially though, you can never move into a group using MQGMO_LOGICAL_ORDER where the first item is not present.

Grouping logical messages:

There are two main reasons for using logical messages in a group:

- You might need to process the messages in a particular order
- You might need to process each message in a group in a related way.

In either case, retrieve the entire group with the same getting application instance.

For example, assume that the group consists of four logical messages. The putting application looks like this:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
```

```
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP

MQCMIT
```

The getting application chooses not to start processing any group until all the messages within it have arrived. specify MQGMO_ALL_MSGS_AVAILABLE for the first message in the group; the option is ignored for subsequent messages within the group.

Once the first logical message of the group is retrieved, use MQGMO_LOGICAL_ORDER to ensure that the remaining logical messages of the group are retrieved in order.

So, the getting application looks like this:

```
/* Wait for the first message in a group, or a message not in a group */
GMO.Options = MQGMO_SYNCPOINT | MQGMO_WAIT
            | MQGMO_ALL_MSGS_AVAILABLE | MQGMO_LOGICAL_ORDER
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
   MQGET
   /* Process each remaining message in the group */
   ...

MQCMIT
```

For further examples of grouping messages, see “Application segmentation of logical messages” on page 146 and “Putting and getting a group that spans units of work.”

Putting and getting a group that spans units of work:

In the previous case, messages or segments cannot start to leave the node (if its destination is remote) or start to be retrieved until the whole group has been put and the unit of work is committed. This might not be what you want if it takes a long time to put the whole group, or if queue space is limited on the node. To overcome this, put the group in several units of work.

If the group is put within multiple units of work, it is possible for some of the group to commit even when the putting application fails. The application must therefore save status information, committed with each unit of work, which it can use after a restart to resume an incomplete group. The simplest place to record this information is in a STATUS queue. If a complete group has been successfully put, the STATUS queue is empty.

If segmentation is involved, the logic is similar. In this case, the StatusInfo must include the *Offset*.

Here is an example of putting the group in several units of work:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

/* First UOW */

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
StatusInfo = GroupId,MsgSeqNumber from MQMD
MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
MQCMIT
```

```
/* Next and subsequent UOWs */
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
StatusInfo = GroupId,MsgSeqNumber from MQMD
MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
MQCMIT

/* Last UOW */
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
MQCMIT
```

If all the units of work have been committed, the entire group has been put successfully, and the STATUS queue is empty. If not, the group must be resumed at the point indicated by the status information. MQPMO_LOGICAL_ORDER cannot be used for the first put, but can thereafter.

Restart processing looks like this:

```
MQGET (StatusInfo from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
if (Reason == MQRC_NO_MSG_AVAILABLE)
   /* Proceed to normal processing */
   ...

else
   /* Group was terminated prematurely */
   Set GroupId, MsgSeqNumber in MQMD to values from Status message
   PMO.Options = MQPMO_SYNCPOINT
   MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP

   /* Now normal processing is resumed.
      Assume this is not the last message */
   PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT
   MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
   MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
   StatusInfo = GroupId,MsgSeqNumber from MQMD
   MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
   MQCMIT
```

From the getting application, you might want to start processing the messages in a group before the whole group has arrived. This improves response times on the messages within the group, and also means that storage is not required for the entire group.

For recovery reasons, you must retrieve each message within a unit of work. However, in order to realize the above benefits, use several units of work for each group of messages.

As with the corresponding putting application, this requires status information to be recorded somewhere automatically as each unit of work is committed. Again, the simplest place to record this information is on a STATUS queue. If a complete group has been successfully processed, the STATUS queue is empty.

Note: For intermediate units of work, you can avoid the MQGET calls from the STATUS queue by specifying that each MQPUT to the status queue is a segment of a message (that is, by setting the MQMF_SEGMENT flag), instead of putting a complete new message for each unit of work. In the last unit of work, a final

segment is put to the status queue specifying MQMF_LAST_SEGMENT, and then the status information is cleared with an MQGET specifying MQGMO_COMPLETE_MSG.

During restart processing, instead of using a single MQGET to get a possible status message, browse the status queue with MQGMO_LOGICAL_ORDER until you reach the last segment (that is, until no further segments are returned). In the first unit of work after restart, also specify the offset explicitly when putting the status segment.

In the following example, we consider only messages within a group, assuming that the application's buffer is always large enough to hold the entire message, whether or not the message has been segmented. MQGMO_COMPLETE_MSG is therefore specified on each MQGET. The same principles apply if segmentation is involved (in this case, the StatusInfo must include the *Offset*).

For simplicity, we assume that a maximum of 4 messages are retrieved within a single UOW:

```
msgs = 0    /* Counts messages retrieved within UOW */
/* Should be no status message at this point */

/* Retrieve remaining messages in the group */
do while ( GroupStatus == MQGS_MSG_IN_GROUP )

   /* Process up to 4 messages in the group */
   GMO.Options = MQGMO_SYNCPOINT | MQGMO_WAIT
               | MQGMO_LOGICAL_ORDER
   do while ( (GroupStatus == MQGS_MSG_IN_GROUP) && (msgs < 4) )
      MQGET
      msgs = msgs + 1
      /* Process this message */
      ...
   /* end while

   /* Have retrieved last message or 4 messages  */
   /* Update status message if not last in group */
   MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
   if ( GroupStatus == MQGS_MSG_IN_GROUP )
      StatusInfo = GroupId,MsgSeqNumber from MQMD
      MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
   MQCMIT
   msgs = 0
/* end while

if ( msgs > 0 )
   /* Come here if there was only 1 message in the group */
   MQCMIT
```

If all the units of work have been committed, the entire group has been retrieved successfully, and the STATUS queue is empty. If not, the group must be resumed at the point indicated by the status information. MQGMO_LOGICAL_ORDER cannot be used for the first retrieve, but can thereafter.

Restart processing looks like this:

```
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
if (Reason == MQRC_NO_MSG_AVAILABLE)
   /* Proceed to normal processing */
   ...

else
   /* Group was terminated prematurely */
```

```
/* The next message on the group must be retrieved by matching
   the sequence number and group id with those retrieved from the
   status information. */
GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT | MQGMO_WAIT
MQGET GMO.MatchOptions = MQMO_MATCH_GROUP_ID | MQMO_MATCH_MSG_SEQ_NUMBER,
      MQMD.GroupId      = value from Status message,
      MQMD.MsgSeqNumber = value from Status message plus 1
msgs = 1
/* Process this message */
...

/* Now normal processing is resumed */
/* Retrieve remaining messages in the group */
do while ( GroupStatus == MQGS_MSG_IN_GROUP )

   /* Process up to 4 messages in the group */
   GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT | MQGMO_WAIT
               | MQGMO_LOGICAL_ORDER
   do while ( (GroupStatus == MQGS_MSG_IN_GROUP) && (msgs < 4) )
      MQGET
      msgs = msgs + 1
      /* Process this message */
      ...

   /* Have retrieved last message or 4 messages  */
   /* Update status message if not last in group */
   MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
   if ( GroupStatus == MQGS_MSG_IN_GROUP )
      StatusInfo = GroupId,MsgSeqNumber from MQMD
      MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
   MQCMIT
   msgs = 0
```

Getting a particular message

There are a number of ways of getting a particular message from a queue. These are: selecting on the MsgId and CorrelId, selecting on the GroupId, MsgSeqNumber and Offset, and selecting on the MsgToken. You can also use a selection string when you open the queue.

To get a particular message from a queue, use the *MsgId* and *CorrelId* fields of the MQMD structure. However, applications can explicitly set these fields, so the values that you specify might not identify a unique message. Table 7 shows which message is retrieved for the possible settings of these fields. These fields are ignored on input if you specify MQGMO_MSG_UNDER_CURSOR in the *GetMsgOpts* parameter of the MQGET call.

Table 7. Using message and correlation identifiers

To retrieve ...	*MsgId*	*CorrelId*
First message in the queue	MQMI_NONE	MQCI_NONE
First message that matches *MsgId*	Nonzero	MQCI_NONE
First message that matches *CorrelId*	MQMI_NONE	Nonzero
First message that matches both *MsgId* and *CorrelId*	Nonzero	Nonzero

In each case, *first* means the first message that satisfies the selection criteria (unless MQGMO_BROWSE_NEXT is specified, when it means the *next* message in the sequence satisfying the selection criteria).

On return, the MQGET call sets the *MsgId* and *CorrelId* fields to the message and correlation identifiers (respectively) of the message returned (if any).

If you set the *Version* field of the MQMD structure to 2, you can use the *GroupId*, *MsgSeqNumber*, and *Offset* fields. Table 8 shows which message is retrieved for the possible settings of these fields.

Table 8. Using the group identifier

To retrieve ...	Match options
First message in the queue	MQMO_NONE
First message that matches *MsgId*	MQMO_MATCH_MSG_ID
First message that matches *CorrelId*	MQMO_MATCH_CORREL_ID
First message that matches *GroupId*	MQMO_MATCH_GROUP_ID
First message that matches *MsgSeqNumber*	MQMO_MATCH_MSG_SEQ_NUMBER
First message that matches *MsgToken*	MQMO_MATCH_MSG_TOKEN
First message that matches *Offset*	MQMO_MATCH_OFFSET

Notes:

1. MQMO_MATCH_XXX implies that the *XXX* field in the MQMD structure is set to the value to be matched.
2. The MQMO flags can be used in combination. For example, MQMO_MATCH_GROUP_ID, MQMO_MATCH_MSG_SEQ_NUMBER, and MQMO_MATCH_OFFSET can be used together to give the segment identified by the *GroupId*, *MsgSeqNumber*, and *Offset* fields.
3. If you specify MQGMO_LOGICAL_ORDER, the message that you are trying to retrieve is affected because the option depends on state information controlled for the queue handle. For information about this, see "Logical and physical ordering" on page 131 and the *WebSphere MQ Application Programming Reference*.

The MQGET call usually retrieves the first message from a queue. If you specify a particular message when you use the MQGET call, the queue manager has to search the queue until it finds that message. This can affect the performance of your application.

If you are using Version 2 or later of the MQGMO structure and do not specify the MQMO_MATCH_MSG_ID or MQMO_MATCH_CORREL_ID flags, you do not need to reset the *MsgId* or *CorrelId* fields respectively between MQGETs.

On WebSphere MQ for z/OS the queue attribute *IndexType* can be used to increase the speed of MQGET operations on the queue. For more information, see "Type of index" on page 142.

You can get a specific message from a queue by specifying its MsgToken and the MatchOption MQMO_MATCH_MSG_TOKEN in the MQGMO structure. The MsgToken is returned by the MQPUT call that originally put that message on the queue, or by previous MQGET operations and remains constant unless the queue manager is restarted.

If you are interested in only a subset of messages on the queue, you can specify which messages you want to process by using a selection string with the MQOPEN or MQSUB call. MQGET then retrieves the next message that satisfies that selection string. For more information about selection strings, see "Selectors" on page 32.

Improving performance of non-persistent messages

When a client requires a message from a server, it sends a request to the server. It sends a separate request for each of the messages it consumes. To improve the performance of a client consuming non persistent messages by avoiding having to send these request messages, a client can be configured to use *read ahead*. Read ahead allows messages to be sent to a client without an application having to request them.

When read ahead is enabled, messages are sent to an in memory buffer on the client called the *read ahead buffer*. The client will have a read ahead buffer for each queue it has open with read ahead enabled. The messages in the read ahead buffer are not persisted. The client periodically updates the server with information about the amount of data it has consumed.

If a client application is restarted, messages in the read ahead buffer can be lost.

Using read ahead can improve performance when consuming non persistent messages from a client application. This performance improvement is available to both MQI and JMS applications. Client applications using MQGET or asynchronous consumption will benefit from the performance improvements when consuming non-persistent messages.

Not all client application designs are suited to using read ahead as not all options are supported for use with read ahead and some options are required to be consistent between MQGET calls when read ahead is enabled. If a client alters its selection criteria between MQGET calls, messages being stored in the read ahead buffer will remain stranded in the client read ahead buffer.

If a backlog of stranded messages with the previous selection criteria are no longer required, a configurable purge interval can be set on the client to automatically purge these messages from the client. The purge interval is one of a group of read ahead tuning options determined by the client. It is possible to tune these options to meet your requirements.

Read ahead is only performed for client bindings. The attribute is ignored for all other bindings.

Read ahead has no impact on triggering. No trigger message is generated when a message is read ahead by the client. Read ahead does not generate accounting and statistics information when it is enabled.

Using read ahead with publish subscribe messaging

When a subscribing application specifies a destination queue to which publications are sent, the DEFREADA value of the specified queue is used as the default read ahead value.

When a subscribing application requests that WebSphere MQ manages the destination to which publications are sent, a managed queue is created as a dynamic queue based upon a predefined model queue. It is the DEFREADA value of the model queue that is used as the default read ahead value. The default model queues SYSTEM.DURABLE.PUBLICATIONS.MODEL or SYSTEM.NONDURABLE.PUBLICATIONS.MODEL are used unless a model queue is defined for this or a parent topic.

MQGET options and read ahead

Not all MQGET options are supported when read ahead is enabled and some options are required to be consistent between MQGET calls when read ahead is enabled.

The following table indicates which options are supported for use with read ahead and whether they can be altered between MQGET calls.

Table 9. MQGET options and read ahead

	Permitted when read ahead is enabled and can be altered between MQGET calls	**Permitted when read ahead is enabled but cannot be altered between MQGET calls[1]**	**MQGET Options that are not permitted when read ahead is enabled[2]**
MQGET MQMD values	MsgId[3] CorrelId[3]	Encoding CodedCharSetId	
MQGET MQGMO Options	• MQGMO_NO_WAIT • MQGMO_BROWSE_MESSAGE _UNDER_CURSOR • MQGMO_BROWSE_FIRST • MQGMO_BROWSE_NEXT • MQGMO_FAIL_IF_QUIESCING	• MQGMO_SYNCPOINT_IF _PERSISTENT • MQGMO_NO_SYNCPOINT • MQGMO_ACCEPT_TRUNCATED _MSG • MQGMO_CONVERT	• MQGMO_SET_SIGNAL • MQGMO_SYNCPOINT • MQGMO_MARK_SKIP _BACKOUT • MQGMO_MSG_UNDER _CURSOR[4] • MQGMO_LOCK • MQGMO_UNLOCK • MQGMO_LOGICAL_ORDER • MQGMO_COMPLETE_MSG • MQGMO_ALL_MSGS_AVAILABLE • MQGMO_ALL_SEGMENTS_ AVAILABLE

Notes:

1. If these options are altered between MQGET calls an MQRC_OPTIONS_CHANGED reason code will be returned.
2. If these options are specified on the first MQGET call then read ahead will be disabled. If these options are specified on a subsequent MQGET call a reason code MQRC_OPTIONS_ERROR will be returned.
3. The client applications needs to be aware that if the MsgId and CorrelId values are altered between MQGET calls messages with the previous values may have already been sent to the client and will remain in the client read ahead buffer until consumed (or automatically purged).
4. MQGMO_MSG_UNDER_CURSOR is not possible with read ahead. Read ahead is disabled when both MQOO_BROWSE and one of the MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE options are specified when opening the queue.

If a client alters its selection criteria between MQGET calls, messages being stored in the read ahead buffer that match the initial selection criteria will not be consumed by the client application and remain stranded in the client read ahead buffer. In situations where the client read ahead buffer contains a large number of stranded messages the benefits associated with read ahead will be lost and a separate request to the server required for each message consumed. To determine whether read ahead is being used efficiently you can use the connection status parameter, READA.

Read ahead can be inhibited when requested by an application due to incompatible options specified on the first MQGET call. In this situation the connection status will show read ahead as being inhibited.

If you decide that because of these restrictions on MQGET, that a client application design is not suited to read ahead, specify the MQOPEN option MQOO_READ_AHEAD_NO. Alternatively set the default read ahead value of the queue being opened altered to either NO or DISABLED.

Enabling and disabling read ahead

By default read ahead is disabled. You can enable read ahead at queue or application level.

To enable read ahead:

- To configure read ahead at the queue level set the queue attribute, DEFREADA to YES.
- To configure read ahead at the application level:
  - to use read ahead wherever possible use the MQOO_READ_AHEAD option on the MQOPEN function call. It will not be possible for the client application to use read ahead if the DEFREADA queue attribute has been set to DISABLED.
  - to use read ahead only when read ahead is enabled on a queue, use the MQOO_READ_AHEAD_AS_Q_DEF option on the MQOPEN function call.

If a client application design is not suited to read ahead you can disable it:

- at the queue level by setting the queue attribute, DEFREADA to NO if you do not want read ahead to be used unless it is requested by a client application, or DISABLED if you do not want read ahead to be used regardless of whether read ahead is required by a client application.
- at the application level by using the MQOO_NO_READ_AHEAD option on the MQOPEN function call.

Two MQCLOSE options allow you to configure what happens to any messages that are being stored in the read ahead buffer if the queue is closed.

- Use MQCO_IMMEDIATE to discard messages in the read ahead buffer.
- Use MQCO_QUIESCE to ensure that messages in the read ahead buffer are consumed by the application before the queue is closed. When MQCLOSE with the MQCO_QUIESCE is issued and there are messages remaining on the read ahead buffer, MQRC_READ_AHEAD_MSGS will be returned with MQCC_WARNING.

Type of index

Supported only on WebSphere MQ for z/OS.

The queue attribute, *IndexType*, specifies the type of index that the queue manager maintains to increase the speed of MQGET operations on the queue.

You have five options:

Value	Description
NONE	No index is maintained. Use this when retrieving messages sequentially (see "Priority" on page 131).
GROUPID	An index of group identifiers is maintained. You *must* use this index type if you want logical ordering of message groups (see "Logical and physical ordering" on page 131).
MSGID	An index of message identifiers is maintained. Use this when retrieving messages using the *MsgId* field as a selection criterion on the MQGET call (see "Getting a particular message" on page 138).
MSGTOKEN	An index of message tokens is maintained.
CORRELID	An index of correlation identifiers is maintained. Use this when retrieving messages using the *CorrelId* field as a selection criterion on the MQGET call (see "Getting a particular message" on page 138).

Note:

1. If you are indexing using the MSGID option or CORRELID option, set the relative *MsgId* or *CorrelId* parameters in the MQMD. It is *not* beneficial to set both.
2. Browse uses the index mechanism to find a message if a queue matches all the following conditions:
   - It has index type MSGID, CORRELID, or GROUPID
   - It is browsed with the same type of id
   - It has messages of only one priority
3. Avoid queues (indexed by *MsgId* or *CorrelId*) containing thousands of messages because this affects restart time. (This does not apply to nonpersistent messages as they are deleted at restart.)
4. MSGTOKEN is used to define queues managed by the z/OS workload manager.

For a full description of the *IndexType* attribute, see the *WebSphere MQ Application Programming Reference*. For conditions needed to change the *IndexType* attribute, see the *WebSphere MQ Script (MQSC) Command Reference*.

Handling messages greater than 4 MB long

Messages can be too large for the application, queue, or queue manager. Depending on the environment, WebSphere MQ provides a number of ways of dealing with messages that are longer than 4 MB.

You can increase the *MaxMsgLength* attribute up to 100 MB on all WebSphere MQ systems at V6 or later. Set this value to reflect the size of the messages using the queue. On WebSphere MQ systems other than WebSphere MQ for z/OS, you can also:

1. Use segmented messages. (Messages can be segmented by either the application or the queue manager.)
2. Use reference messages.

Each of these approaches is described in the remainder of this section.

Increasing the maximum message length

The *MaxMsgLength* queue manager attribute defines the maximum length of a message that can be handled by a queue manager. Similarly, the *MaxMsgLength* queue attribute is the maximum length of a message that can be handled by a queue. The *default* maximum message length supported depends on the environment in which you are working.

If you are handling large messages, you can alter these attributes independently. You can set the queue manager attribute value between 32768 bytes and 100 MB; you can set the queue attribute value between 0 and 100 MB.

After changing one or both of the *MaxMsgLength* attributes, restart your applications and channels to ensure that the changes take effect.

When these changes are made, the message length must be less than or equal to both the queue and the queue manager *MaxMsgLength* attributes. However, existing messages might be longer than either attribute.

If the message is too big for the queue, MQRC_MSG_TOO_BIG_FOR_Q is returned. Similarly, if the message is too big for the queue manager, MQRC_MSG_TOO_BIG_FOR_Q_MGR is returned.

This method of handling large messages is easy and convenient. However, consider the following factors before using it:

- Uniformity among queue managers is reduced. The maximum size of message data is determined by the *MaxMsgLength* for each queue (including transmission queues) on which the message will be put. This value is often defaulted to the queue manager's *MaxMsgLength*, especially for transmission queues. This makes it difficult to predict whether a message is too large when it is to travel to a remote queue manager.
- Usage of system resources is increased. For example, applications need larger buffers, and on some platforms, there might be increased usage of shared storage. Queue storage should be affected only if actually required for larger messages.
- Channel batching is affected. A large message still counts as just one message towards the batch count but needs longer to transmit, thereby increasing response times for other messages.

Message segmentation

Not supported in WebSphere MQ for z/OS.

Increasing the maximum message length as discussed in topic "Increasing the maximum message length" on page 143 has some negative implications. Also, it can still result in the message being too large for the queue or queue manager. In these cases, you can segment a message. For information about segments, see "Message groups" on page 43.

The next sections look at common uses for segmenting messages. For putting and destructively getting, it is assumed that the MQPUT or MQGET calls *always* operate within a unit of work. We strongly recommend that you always use this technique, to reduce the possibility of incomplete groups being present in the network. Single-phase commit by the queue manager is assumed, but of course other coordination techniques are equally valid.

Also, in the getting applications, it is assumed that if multiple servers are processing the same queue, each server executes similar code, so that one server never fails to find a message or segment that it expects to be there (because it had specified MQGMO_ALL_MSGS_AVAILABLE or MQGMO_ALL_SEGMENTS_AVAILABLE earlier).

Segmentation and reassembly by queue manager:

This is the simplest scenario, in which one application puts a message to be retrieved by another. The message might be large: not too large for either the putting or the getting application to handle in a single buffer, but too large for the queue manager or a queue on which the message is to be put.

The only changes necessary for these applications are for the putting application to authorize the queue manager to perform segmentation if necessary:

```
PMO.Options = (existing options)
MQPUT MD.MsgFlags = MQMF_SEGMENTATION_ALLOWED
```

and for the getting application to ask the queue manager to reassemble the message if it has been segmented:

```
GMO.Options = MQGMO_COMPLETE_MSG | (existing options)
MQGET
```

The application buffer must be large enough to contain the reassembled message (unless you include the MQGMO_ACCEPT_TRUNCATED_MSG option).

If data conversion is necessary, the getting application might have to do it by specifying MQGMO_CONVERT. This should be straightforward because the data conversion exit is presented with the complete message. Do not attempt to convert data in a sender channel if the message is segmented, and the format of the data is such that the data-conversion exit cannot carry out the conversion on incomplete data.

Application segmentation:

Application segmentation is used for two main reasons:

1. Queue-manager segmentation alone is not adequate because the message is too large to be handled in a single buffer by the applications.
2. Data conversion must be performed by sender channels, and the format is such that the putting application needs to stipulate where the segment boundaries are to be in order for conversion of an individual segment to be possible.

However, if data conversion is not an issue, or if the getting application always uses MQGMO_COMPLETE_MSG, queue-manager segmentation can also be allowed by specifying MQMF_SEGMENTATION_ALLOWED. In our example, the application segments the message into four segments:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_SEGMENT

MQCMIT
```

If you do not use MQPMO_LOGICAL_ORDER, the application must set the *Offset* and the length of each segment. In this case, logical state is not maintained automatically.

The getting application cannot guarantee to have a buffer large enough to hold any reassembled message. It must therefore be prepared to process segments individually.

For messages that are segmented, this application does not want to start processing one segment until all the segments that constitute the logical message are present. MQGMO_ALL_SEGMENTS_AVAILABLE is therefore specified for the first segment. If you specify MQGMO_LOGICAL_ORDER and there is a current logical message, MQGMO_ALL_SEGMENTS_AVAILABLE is ignored.

Once the first segment of a logical message has been retrieved, use MQGMO_LOGICAL_ORDER to ensure that the remaining segments of the logical message are retrieved in order.

No consideration is given to messages within different groups. If such messages occur, they are processed in the order in which the first segment of each message appears on the queue.

```
GMO.Options = MQGMO_SYNCPOINT | MQGMO_LOGICAL_ORDER
            | MQGMO_ALL_SEGMENTS_AVAILABLE | MQGMO_WAIT
do while ( SegmentStatus == MQSS_SEGMENT )
   MQGET
   /* Process each remaining segment of the logical message */
   ...

MQCMIT
```

Application segmentation of logical messages:

The messages must be maintained in logical order in a group, and some or all of them might be so large that they require application segmentation.

In our example, a group of four logical messages is to be put. All but the third message are large, and require segmentation, which is performed by the putting application:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_LAST_SEGMENT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_LAST_SEGMENT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP

MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_LAST_SEGMENT

MQCMIT
```

In the getting application, MQGMO_ALL_MSGS_AVAILABLE is specified on the first MQGET. This means that no messages or segments of a group are retrieved until the entire group is available. When the first physical message of a group has been retrieved, MQGMO_LOGICAL_ORDER is used to ensure that the segments and messages of the group are retrieved in order:

```
GMO.Options = MQGMO_SYNCPOINT | MQGMO_LOGICAL_ORDER
            | MQGMO_ALL_MESSAGES_AVAILABLE | MQGMO_WAIT

do while ( (GroupStatus   != MQGS_LAST_MSG_IN_GROUP) ||
             (SegmentStatus != MQGS_LAST_SEGMENT) )
   MQGET
   /* Process a segment or complete logical message.  Use the GroupStatus
      and SegmentStatus information to see what has been returned */
   ...

MQCMIT
```

Note: If you specify MQGMO_LOGICAL_ORDER and there is a current group, MQGMO_ALL_MSGS_AVAILABLE is ignored.

Putting and getting a segmented message that spans units of work:

You can put and get a segmented message that spans a unit of work in a similar way to "Putting and getting a group that spans units of work" on page 135.

You cannot, however, put or get segmented messages in a global unit of work.

Reference messages

Not supported in WebSphere MQ for z/OS.

This method allows a large object to be transferred from one node to another without storing the object on WebSphere MQ queues at either the source or the destination nodes. This is of particular benefit when the data already exists in another form, for example, for mail applications.

To do this, you specify a message exit at both ends of a channel. For information on how to do this, see *WebSphere MQ Intercommunications*.

WebSphere MQ defines the format of a reference message header (MQRMH). See the *WebSphere MQ Application Programming Reference* for a description of this. This is recognized by means of a defined format name and might be followed by actual data.

To initiate transfer of a large object, an application can put a message consisting of a reference message header with no data following it. As this message leaves the node, the message exit retrieves the object in an appropriate way and appends it to the reference message. It then returns the message (now larger than before) to the sending Message Channel Agent for transmission to the receiving MCA.

Another message exit is configured at the receiving MCA. When this message exit sees one of these messages, it creates the object using the object data that was appended and passes on the reference message *without* it. The reference message can now be received by an application and this application knows that the object (or at least the portion of it represented by this reference message) has been created at this node.

The maximum amount of object data that a sending message exit can append to the reference message is limited by the negotiated maximum message length for the channel. The exit can return only a single message to the MCA for each message that it is passed, so the putting application can put several messages to cause one object to be transferred. Each message must identify the *logical* length and offset of the object that is to be appended to it. However, in cases where it is not possible to know the total size of the object or the maximum size allowed by the channel, design the sending message exit so that the putting application just puts a single message, and the exit itself puts the next message on the transmission queue when it has appended as much data as it can to the message it has been passed.

Before using this method of dealing with large messages, consider the following:

- The MCA and the message exit run under a WebSphere MQ user ID. The message exit (and therefore, the user ID) needs to access the object to either retrieve it at the sending end or create it at the receiving end; this might only be feasible in cases where the object is universally accessible. This raises a security issue.
- If the reference message with bulk data appended to it must travel through several queue managers before reaching its destination, the bulk data *is* present on WebSphere MQ queues at the intervening nodes. However, no special support or exits need to be provided in these cases.

- Designing your message exit is made difficult if rerouting or dead-letter queuing is allowed. In these cases, the portions of the object might arrive out of order.
- When a reference message arrives at its destination, the receiving message exit creates the object. However, this is not synchronized with the MCA's unit of work, so if the batch is backed out, another reference message containing this same portion of the object will arrive in a later batch, and the message exit might attempt to re-create the same portion of the object. If the object is, for example, a series of database updates, this might be unacceptable. If so, the message exit must keep a log of which updates have been applied; this might require the use of a WebSphere MQ queue.
- Depending on the characteristics of the object type, the message exits and applications might need to cooperate in maintaining use counts, so that the object can be deleted when it is no longer needed. An instance identifier might also be required; a field is provided for this in the reference message header (see the *WebSphere MQ Application Programming Reference*).
- If a reference message is put as a distribution list, the object must be retrievable for each resulting distribution list or individual destination at that node. You might need to maintain use counts. Also consider the possibility that a given node might be the final node for some of the destinations in the list, but an intermediate node for others.
- Bulk data is not ususally converted. This is because conversion takes place *before* the message exit is invoked. For this reason, conversion must not be requested on the originating sender channel. If the reference message passes through an intermediate node, the bulk data is converted when sent from the intermediate node, if requested.
- Reference messages cannot be segmented.

Using the MQRMH and MQMD structures:

See the *WebSphere MQ Application Programming Reference* for a description of the fields in the reference message header and the message descriptor.

In the MQMD structure, set the *Format* field to MQFMT_REF_MSG_HEADER. The MQHREF format, when requested on MQGET, is converted automatically by WebSphere MQ along with any bulk data that follows.

Here is an example of the use of the *DataLogicalOffset* and *DataLogicalLength* fields of the MQRMH:

A putting application might put a reference message with:

- No physical data
- *DataLogicalLength* = 0 (this message represents the entire object)
- *DataLogicalOffset* = 0.

Assuming that the object is 70 000 bytes long, the sending message exit sends the first 40 000 bytes along the channel in a reference message containing:

- 40 000 bytes of physical data following the MQRMH
- *DataLogicalLength* = 40000
- *DataLogicalOffset* = 0 (from the start of the object).

It then places another message on the transmission queue containing:

- No physical data

- *DataLogicalLength* = 0 (to the end of the object). You could specify a value of 30 000 here.
- *DataLogicalOffset* = 40000 (starting from this point).

When this message exit is seen by the sending message exit, the remaining 30,000 bytes of data is appended, and the fields are set to:

- 30,000 bytes of physical data following the MQRMH
- *DataLogicalLength* = 30000
- *DataLogicalOffset* = 40000 (starting from this point).

The MQRMHF_LAST flag is also set.

For a description of the sample programs provided for the use of reference messages, see "Sample programs (all platforms except z/OS)" on page 395.

Waiting for messages

If you want a program to wait until a message arrives on a queue, specify the MQGMO_WAIT option in the *Options* field of the MQGMO structure. Use the *WaitInterval* field of the MQGMO structure to specify the maximum time (in milliseconds) that you want an MQGET call to wait for a message to arrive on a queue.

If the message does not arrive within this time, the MQGET call completes with the MQRC_NO_MSG_AVAILABLE reason code.

You can specify an unlimited wait interval using the constant MQWI_UNLIMITED in the *WaitInterval* field. However, events outside your control could cause your program to wait for a long time, so use this constant with caution. IMS applications must not specify an unlimited wait interval because this would prevent the IMS system terminating. (When IMS terminates, it requires all dependent regions to end.) Instead, IMS applications can specify a finite wait interval; then, if the call completes without retrieving a message after that interval, issue another MQGET call with the wait option.

Note: If more than one program is waiting on the same shared queue to *remove* a message, only one program is activated by a message arriving. However, if more than one program is waiting to browse a message, all the programs can be activated. For more information, see the description of the *Options* field of the MQGMO structure in the *WebSphere MQ Application Programming Reference*.

If the state of the queue or the queue manager changes before the wait interval expires, the following actions occur:

- If the queue manager enters the quiescing state, and you used the MQGMO_FAIL_IF_QUIESCING option, the wait is canceled and the MQGET call completes with the MQRC_Q_MGR_QUIESCING reason code. Without this option, the call remains waiting.
- On z/OS, if the connection (for a CICS or IMS application) enters the quiescing state, and you used the MQGMO_FAIL_IF_QUIESCING option, the wait is canceled and the MQGET call completes with the MQRC_CONN_QUIESCING reason code. Without this option, the call remains waiting.
- If the queue manager is forced to stop, or is canceled, the MQGET call completes with either the MQRC_Q_MGR_STOPPING or the MQRC_CONNECTION_BROKEN reason code.

- If the attributes of the queue (or a queue to which the queue name resolves) are changed so that get requests are now inhibited, the wait is canceled and the MQGET call completes with the MQRC_GET_INHIBITED reason code.
- If the attributes of the queue (or a queue to which the queue name resolves) are changed in such a way that the FORCE option is required, the wait is canceled and the MQGET call completes with the MQRC_OBJECT_CHANGED reason code.

If you want your application to wait on more than one queue, use the signal facility of WebSphere MQ for z/OS (see "Signaling"). For more information about the circumstances in which these actions occur, see the *WebSphere MQ Application Programming Reference*.

Signaling

Signaling is supported only on WebSphere MQ for z/OS.

Signaling is an option on the MQGET call to allow the operating system to notify (or *signal*) a program when an expected message arrives on a queue. This is similar to the *get with wait* function described in topic "Waiting for messages" on page 149 because it allows your program to continue with other work while waiting for the signal. However, if you use signaling, you can free the application thread and rely on the operating system to notify the program when a message arrives.

To set a signal

To set a signal, do the following in the MQGMO structure that you use on your MQGET call:

1. Set the MQGMO_SET_SIGNAL option in the *Options* field.
2. Set the maximum life of the signal in the *WaitInterval* field. This sets the length of time (in milliseconds) for which you want WebSphere MQ to monitor the queue. Use the MQWI_UNLIMITED value to specify an unlimited life.

   Note: IMS applications must not specify an unlimited wait interval because this would prevent the IMS system from terminating. (When IMS terminates, it requires all dependent regions to end.) Instead, IMS applications can examine the state of the ECB at regular intervals (see step 3). A program can have signals set on several queue handles at the same time:
3. Specify the address of the *Event Control Block* (ECB) in the *Signal1* field. This notifies you of the result of your signal. The ECB storage must remain available until the queue is closed.

Note: You cannot use the MQGMO_SET_SIGNAL option in conjunction with the MQGMO_WAIT option.

When the message arrives

When a suitable message arrives, a completion code is returned to the ECB.

The completion code describes one of the following:

- The message that you set the signal for has arrived on the queue. The message is not reserved for the program that requested a signal, so the program must issue an MQGET call again to get the message.

  Note: Another application could get the message in the time between your receiving the signal and issuing another MQGET call.

- The wait interval you set has expired and the message you set the signal for did not arrive on the queue. WebSphere MQ has canceled the signal.
- The signal has been canceled. This happens, for example, if the queue manager stops, or the attribute of the queue is changed, so that MQGET calls are no longer allowed.

When a suitable message is already on the queue, the MQGET call completes in the same way as an MQGET call without signaling. Also, if an error is detected immediately, the call completes and the return codes are set.

When the call is accepted and no message is immediately available, control is returned to the program so that it can continue with other work. None of the output fields in the message descriptor are set, but the *CompCode* and *Reason* parameters are set to MQCC_WARNING and MQRC_SIGNAL_REQUEST_ACCEPTED, respectively.

For information on what WebSphere MQ can return to your application when it makes an MQGET call using signaling, see the *WebSphere MQ Application Programming Reference*.

If the program has no other work to do while it is waiting for the ECB to be posted, it can wait for the ECB using:

- For a CICS Transaction Server for OS/390 program, the EXEC CICS WAIT EXTERNAL command
- For batch and IMS programs, the z/OS WAIT macro

If the state of the queue or the queue manager changes while the signal is set (that is, the ECB has not yet been posted), the following actions occur:

- If the queue manager enters the quiescing state, and you used the MQGMO_FAIL_IF_QUIESCING option, the signal is canceled. The ECB is posted with the MQEC_Q_MGR_QUIESCING completion code. Without this option, the signal remains set.
- If the queue manager is forced to stop, or is canceled, the signal is canceled. The signal is delivered with the MQEC_WAIT_CANCELED completion code.
- If the attributes of the queue (or a queue to which the queue name resolves) are changed so that get requests are now inhibited, the signal is canceled. The signal is delivered with the MQEC_WAIT_CANCELED completion code.

Note:

1. If more than one program has set a signal on the same shared queue to remove a message, only one program is activated by a message arriving. However, if more than one program is waiting to browse a message, all the programs can be activated. The rules that the queue manager follows when deciding which applications to activate are the same as those for waiting applications: for more information, see the description of the *Options* field of the MQGMO structure in the *WebSphere MQ Application Programming Reference*.
2. If there is more than one MQGET call waiting for the same message, with a mixture of wait and signal options, each waiting call is considered equally. For more information, see the description of the *Options* field of the MQGMO structure in the *WebSphere MQ Application Programming Reference*.
3. Under some conditions, it is possible both for an MQGET call to retrieve a message and for a signal (resulting from the arrival of the same message) to be

delivered. This means that when your program issues another MQGET call (because the signal was delivered), there could be no message available. Design your program to test for this situation.

For information about how to set a signal, see the description of the MQGMO_SET_SIGNAL option and the *Signal1* field in the *WebSphere MQ Application Programming Reference*.

Skipping backout

Supported only on WebSphere MQ for z/OS.

As part of a unit of work, an application program can issue one or more MQGET calls to get messages from a queue. If the application program detects an error, it can back out the unit of work. This restores all the resources updated during that unit of work to the state that they were in before the unit of work started, and reinstates the messages retrieved by the MQGET calls.

Once reinstated, these messages are available to subsequent MQGET calls issued by the application program. In many cases, this does not cause a problem for the application program. However, in cases where the error leading to the backout cannot be circumvented, having the message reinstated on the queue can cause the application program to enter an *MQGET-error-backout* loop.

To avoid this problem, specify the MQGMO_MARK_SKIP_BACKOUT option on the MQGET call. This marks the MQGET request as not being involved in application-initiated backout; that is, it should not be backed out. Use of this option means that when a backout occurs, updates to other resources are backed out as required, but the marked message is treated as if it had been retrieved under a new unit of work.

The application program must issue a WebSphere MQ call either to commit the new unit of work, or to back out the new unit of work. For example, the program can perform exception handling, such as informing the originator that the message has been discarded, and commit the unit of work so removing the message from the queue, If the new unit of work is backed out (for any reason) the message is reinstated on the queue.

Within a unit of work, there can be only one MQGET request marked as skipping backout; however, there can be several other messages that are not marked as skipping backout. Once a message has been marked as skipping backout, any further MQGET calls within the unit of work that specify MQGMO_MARK_SKIP_BACKOUT fail with reason code MQRC_SECOND_MARK_NOT_ALLOWED.

Note:

1. The marked message skips backout only if the unit of work containing it is terminated by an application request to back it out. If the unit of work is backed out for any other reason, the message is backed out onto the queue in the same way that it would be if it was not marked to skip backout.
2. Skip backout is not supported within DB2 stored procedures participating in units of work controlled by RRS. For example, an MQGET call with the MQGMO_MARK_SKIP_BACKOUT option will fail with the reason code MQRC_OPTION_ENVIRONMENT_ERROR.

Figure 16 illustrates a typical sequence of steps that an application program might contain when an MQGET request is required to skip backout.

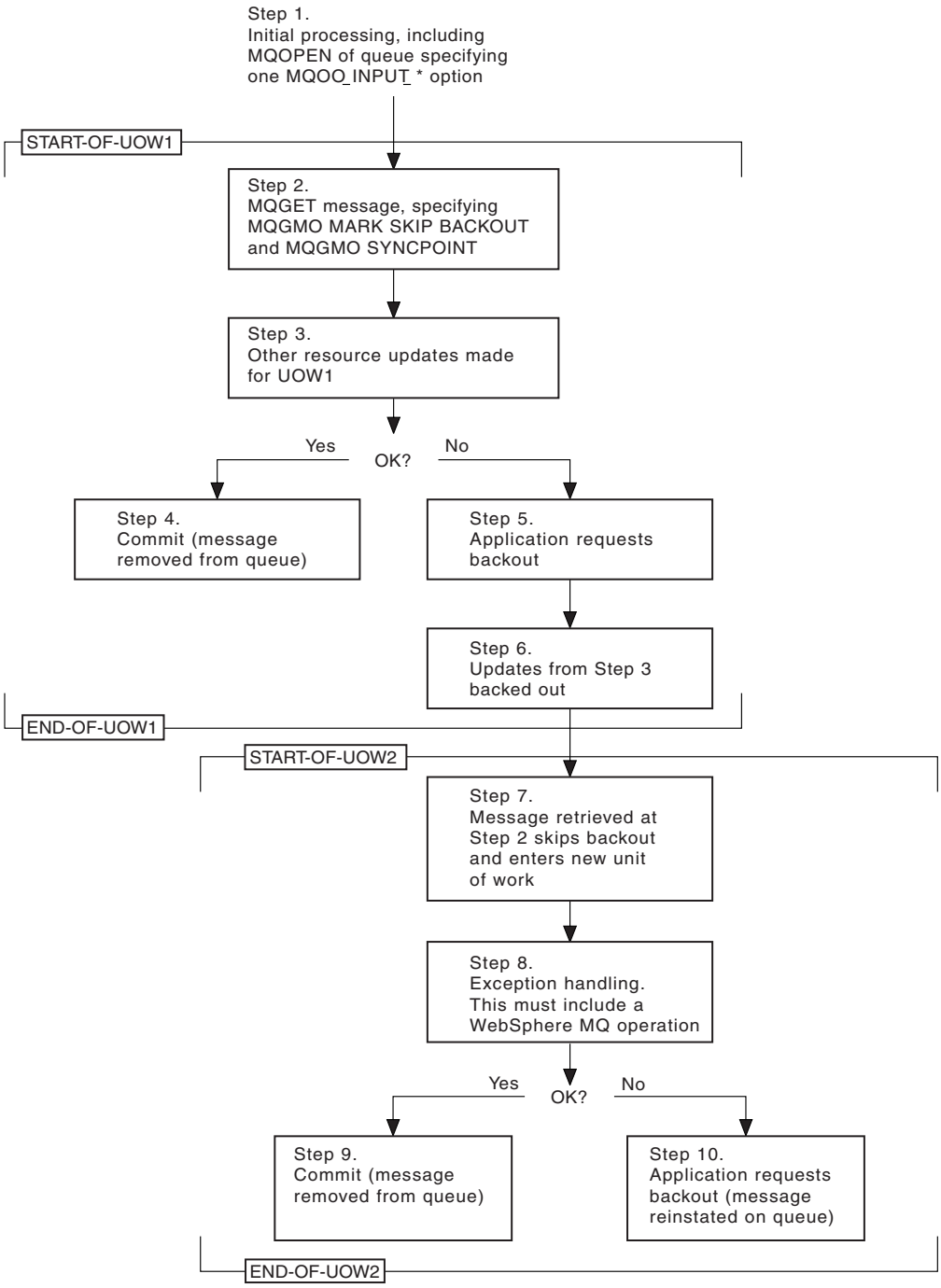


Figure 16. Skipping backout using MQGMO_MARK_SKIP_BACKOUT

The steps in Figure 16 are:

Step 1 Initial processing occurs within the transaction, including an MQOPEN call to open the queue (specifying one of the MQOO_INPUT_* options in order to get messages from the queue in Step 2).

Step 2 MQGET is called, with MQGMO_SYNCPOINT and

MQGMO_MARK_SKIP_BACKOUT. MQGMO_SYNCPOINT is required because MQGET must be within a unit of work for MQGMO_MARK_SKIP_BACKOUT to be effective. In Figure 16 on page 153 this unit of work is referred to as UOW1.

Step 3 Other resource updates are made as part of UOW1. These can include further MQGET calls (issued without MQGMO_MARK_SKIP_BACKOUT).

Step 4 All updates from Steps 2 and 3 complete as required. The application program commits the updates and UOW1 ends. The message retrieved in Step 2 is removed from the queue.

Step 5 Some of the updates from Steps 2 and 3 do not complete as required. The application program requests that the updates made during these steps are backed out.

Step 6 The updates made in Step 3 are backed out.

Step 7 The MQGET request made in Step 2 skips backout and becomes part of a new unit of work, UOW2.

Step 8 UOW2 performs exception handling in response to UOW1 being backed out. (For example, an MQPUT call to another queue, indicating that a problem occurred that caused UOW1 to be backed out.)

Step 9 Step 8 completes as required, the application program commits the activity, and UOW2 ends. As the MQGET request is part of UOW2 (see Step 7), this commit causes the message to be removed from the queue.

Step 10
Step 8 does not complete as required and the application program backs out UOW2. Because the get message request is part of UOW2 (see Step 7), it too is backed out and reinstated on the queue. It is now available to further MQGET calls issued by this or another application program (in the same way as any other message on the queue).

Application data conversion

When necessary, MCAs convert the message descriptor and header data into the required character set and encoding. Either end of the link (that is, the local MCA or the remote MCA) can do the conversion.

When an application puts messages on a queue, the local queue manager adds control information to the message descriptors to facilitate the control of the messages when they are processed by queue managers and MCAs. Depending on the environment, the message header data fields are created in the character set and encoding of the local system.

When you move messages between systems, you sometimes need to convert the application data into the character set and encoding required by the receiving system. This can be done either from within application programs on the receiving system or by the MCAs on the sending system. If data conversion is supported on the receiving system, use application programs to convert the application data, rather than depending on the conversion having already occurred at the sending system.

Application data is converted within an application program when you specify the MQGMO_CONVERT option in the *Options* field of the MQGMO structure passed to an MQGET call, and *all* the following are true:

- The *CodedCharSetId* or *Encoding* fields set in the MQMD structure associated with the message on the queue differ from the *CodedCharSetId* or *Encoding* fields set in the MQMD structure specified on the MQGET call.
- The *Format* field in the MQMD structure associated with the message is not MQFMT_NONE.
- The *BufferLength* specified on the MQGET call is not zero.
- The message data length is not zero.
- The queue manager supports conversion between the *CodedCharSetId* and *Encoding* fields specified in the MQMD structures associated with the message and the MQGET call. See the *WebSphere MQ Application Programming Reference* for details of the coded character set identifiers and machine encodings supported.
- The queue manager supports conversion of the message format. If the *Format* field of the MQMD structure associated with the message is one of the built-in formats, the queue manager can convert the message. If the *Format* is not one of the built-in formats, you need to write a data-conversion exit to convert the message.

If the sending MCA is to convert the data, specify the CONVERT(YES) keyword on the definition of each sender or server channel for which conversion is required. If the data conversion fails, the message is sent to the DLQ at the sending queue manager and the *Feedback* field of the MQDLH structure indicates the reason. If the message cannot be put on the DLQ, the channel closes and the unconverted message remains on the transmission queue. Data conversion within applications rather than at sending MCAs avoids this situation.

As a general rule, data in the message that is described as *character* data by the built-in format or data-conversion exit is converted from the coded character set used by the message to that requested, and *numeric* fields are converted to the encoding requested.

For further details of the conversion processing conventions used when converting the built-in formats, and for information about writing your own data-conversion exits, see "Writing data-conversion exits" on page 163. See also the *WebSphere MQ Application Programming Reference* for information about the language support tables and about the supported machine encodings.

Conversion of EBCDIC newline characters

If you need to ensure that the data that you send from an EBCDIC platform to an ASCII one is identical to the data that you receive back again, you must control the conversion of EBCDIC newline characters.

You can do this using a platform-dependent switch that forces WebSphere MQ to use the unmodified conversion tables, but you must be aware of the inconsistent behavior that might result.

The problem arises because the EBCDIC newline character is not converted consistently across platforms or conversion tables. As a result, if the data is displayed on an ASCII platform, the formatting might be incorrect. This would make it difficult, for example, to administer an i5/OS system remotely from an ASCII platform using RUNMQSC.

See the WebSphere MQ System Administration Guide for further information about converting EBCDIC-format data to ASCII format.

Browsing messages on a queue

To use the MQGET call to browse the messages on a queue:

1. Call MQOPEN to open the queue for browsing, specifying the MQOO_BROWSE option.
2. To browse the first message on the queue, call MQGET with the MQGMO_BROWSE_FIRST option. To find the message that you want, call MQGET repeatedly with the MQGMO_BROWSE_NEXT option to step through many messages.

   You *must* set the *MsgId* and *CorrelId* fields of the MQMD structure to null after each MQGET call in order to see all messages.
3. Call MQCLOSE to close the queue.

The browse cursor

When you open (MQOPEN) a queue for browsing, the call establishes a browse cursor for use with MQGET calls that use one of the browse options. You can think of the browse cursor as a logical pointer that is positioned before the first message on the queue.

You can have more than one browse cursor active (from a single program) by issuing several MQOPEN requests for the same queue.

When you call MQGET for browsing, use one of the following options in your MQGMO structure:

MQGMO_BROWSE_FIRST
Gets a copy of the first message that satisfies the conditions specified in your MQMD structure.

MQGMO_BROWSE_NEXT
Gets a copy of the next message that satisfies the conditions specified in your MQMD structure.

MQGMO_BROWSE_MSG_UNDER_CURSOR
Gets a copy of the message currently pointed to by the cursor, that is, the one that was last retrieved using either the MQGMO_BROWSE_FIRST or the MQGMO_BROWSE_NEXT option.

In all cases, the message remains on the queue.

When you open a queue, the browse cursor is positioned logically just before the first message on the queue. This means that if you make your MQGET call immediately after your MQOPEN call, you can use the MQGMO_BROWSE_NEXT option to browse the first message; you do not have to use the MQGMO_BROWSE_FIRST option.

The order in which messages are copied from the queue is determined by the *MsgDeliverySequence* attribute of the queue. (For more information, see “The order in which messages are retrieved from a queue” on page 131.)

Queues in FIFO (first in, first out) sequence:

The first message in a queue in this sequence is the message that has been on the queue the longest.

Use MQGMO_BROWSE_NEXT to read the messages sequentially in the queue. You will see any messages put to the queue while you are browsing, as a queue in this sequence has messages placed at the end. When the cursor recognizes that it has reached the end of the queue, the browse cursor stays where it is and returns with MQRC_NO_MSG_AVAILABLE. You can then either leave it there waiting for further messages or reset it to the beginning of the queue with a MQGMO_BROWSE_FIRST call.

Queues in priority sequence:

The first message in a queue in this sequence is the message that has been on the queue the longest and that has the highest priority at the time that the MQOPEN call is issued.

Use MQGMO_BROWSE_NEXT to read the messages in the queue.

The browse cursor points to the next message, working from the priority of the first message to finish with the message at the lowest priority. It browses any messages put to the queue during this time as long as they are of priority equal to, or lower than, the message identified by the current browse cursor.

Any messages put to the queue of higher priority can be browsed only by:

- Opening the queue for browse again, at which point a new browse cursor is established
- Using the MQGMO_BROWSE_FIRST option

Uncommitted messages:

An uncommitted message is never visible to a browse; the browse cursor skips past it.

Messages within a unit-of-work cannot be browsed until the unit-of-work is committed. Messages do not change their position on the queue when committed, so skipped, uncommitted messages will not be seen, even when they *are* committed, unless you use the MQGMO_BROWSE_FIRST option and work though the queue again.

Change to queue sequence:

If the message delivery sequence is changed from priority to FIFO while there are messages on the queue, the order of the messages that are already queued is not changed. Messages added to the queue subsequently take the default priority of the queue.

Using the queue's index:

Supported only on WebSphere MQ for z/OS.

When you browse an indexed queue that contains only messages of a single priority (either persistent or nonpersistent or both), the queue manager performs the browse by making use of the index, when any of the following forms of browse are used:

1. If the queue is indexed by MSGID, and the above condition is true, browse requests that pass a MSGID in the MQMD structure are processed using the index to find the target message.

2. If the queue is indexed by CORRELID, and the above condition is true, browse requests that pass a CORRELID in the MQMD structure are processed using the index to find the target message.
3. If the queue is indexed by GROUPID, and the above condition is true, browse requests that pass a GROUPID in the MQMD structure are processed using the index to find the target message.

If the browse request does not pass a MSGID, CORRELID, or GROUPID in the MQMD structure, the queue is indexed, and a message is returned, the index entry for the message must be found, and information within it used to update the browse cursor. If you use a wide selection of index values, this extra processing adds little overhead to the browse request.

Browsing messages when the message length is unknown

To browse a message when you do not know the size of the message, and you do not want to use the *MsgId*, *CorrelId*, or *GroupId* fields to locate the message, you can use the MQGMO_BROWSE_MSG_UNDER_CURSOR option:

1. Issue an MQGET with:
   - Either the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option
   - The MQGMO_ACCEPT_TRUNCATED_MSG option
   - Buffer length zero

   Note: If another program is likely to get the same message, consider using the MQGMO_LOCK option as well. MQRC_TRUNCATED_MSG_ACCEPTED should be returned.
2. Use the returned *DataLength* to allocate the storage needed.
3. Issue an MQGET with the MQGMO_BROWSE_MSG_UNDER_CURSOR.

The message pointed to is the last one that was retrieved; the browse cursor will not have moved. You can choose either to lock the message using the MQGMO_LOCK option, or to unlock a locked message using MQGMO_UNLOCK option.

The call fails if no MQGET with either the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT options has been issued successfully since the queue was opened.

Removing a message that you have browsed

You can remove from the queue a message that you have already browsed provided that you have opened the queue for removing messages as well as for browsing. (You must specify one of the MQOO_INPUT_* options, as well as the MQOO_BROWSE option, on your MQOPEN call.)

To remove the message, call MQGET again, but in the *Options* field of the MQGMO structure, specify MQGMO_MSG_UNDER_CURSOR. In this case, the MQGET call ignores the *MsgId*, *CorrelId*, and *GroupId* fields of the MQMD structure.

In the time between your browsing and removal steps, another program might have removed messages from the queue, including the message under your browse cursor. In this case, your MQGET call returns a reason code to say that the message is not available.

Browsing messages in logical order

"Logical and physical ordering" on page 131 discusses the difference between the logical and physical order of messages on a queue. This distinction is particularly important when browsing a queue, because, in general, messages are not being deleted and browse operations do not necessarily start at the beginning of the queue.

If an application browses through the various messages of one group (using logical order), it is important that logical order should be followed to reach the start of the next group, because the last message of one group might occur physically *after* the first message of the next group. The MQGMO_LOGICAL_ORDER option ensures that logical order is followed when scanning a queue.

Use MQGMO_ALL_MSGS_AVAILABLE (or MQGMO_ALL_SEGMENTS_AVAILABLE) with care for browse operations. Consider the case of logical messages with MQGMO_ALL_MSGS_AVAILABLE. The effect of this is that a logical message is available only if all the remaining messages in the group are also present. If they are not, the message is passed over. This can mean that when the missing messages arrive subsequently, they are not noticed by a browse-next operation.

For example, if the following logical messages are present,

```
Logical message 1 (not last) of group 123
Logical message 1 (not last) of group 456
Logical message 2 (last)     of group 456
```

and a browse function is issued with MQGMO_ALL_MSGS_AVAILABLE, the first logical message of group 456 is returned, leaving the browse cursor on this logical message. If the second (last) message of group 123 now arrives:

```
Logical message 1 (not last) of group 123
Logical message 2 (last)     of group 123
Logical message 1 (not last) of group 456  <=== browse cursor
Logical message 2 (last)     of group 456
```

and the same browse-next function is issued, it is not noticed that group 123 is now complete, because the first message of this group is *before* the browse cursor.

In some cases (for example, if messages are retrieved destructively when the group is present in its entirety), you can use MQGMO_ALL_MSGS_AVAILABLE together with MQGMO_BROWSE_FIRST. Otherwise, you must repeat the browse scan to take note of newly-arrived messages that have been missed; just issuing MQGMO_WAIT together with MQGMO_BROWSE_NEXT and MQGMO_ALL_MSGS_AVAILABLE does not take account of them. (This also happens to higher-priority messages that might arrive after scanning the messages is complete.)

The next sections look at browsing examples that deal with unsegmented messages; segmented messages follow similar principles.

Browsing messages in groups:

In this example, the application browses through each message on the queue, in logical order.

Messages on the queue might be grouped. For grouped messages, the application does not want to start processing any group until all the messages within it have

arrived. MQGMO_ALL_MSGS_AVAILABLE is therefore specified for the first message in the group; for subsequent messages in the group, this option is unnecessary.

MQGMO_WAIT is used in this example. However, although the wait can be satisfied if a new group arrives, for the reasons in "Browsing messages in logical order" on page 159, it is not satisfied if the browse cursor has already passed the first logical message in a group, and the remaining messages now arrive. Nevertheless, waiting for a suitable interval ensures that the application does not constantly loop while waiting for new messages or segments.

MQGMO_LOGICAL_ORDER is used throughout, to ensure that the scan is in logical order. This contrasts with the destructive MQGET example, where because each group is being removed, MQGMO_LOGICAL_ORDER is not used when looking for the first (or only) message in a group.

It is assumed that the application's buffer is always large enough to hold the entire message, whether or not the message has been segmented. MQGMO_COMPLETE_MSG is therefore specified on each MQGET.

The following gives an example of browsing logical messages in a group:

```
/* Browse the first message in a group, or a message not in a group */
GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
            | MQGMO_ALL_MSGS_AVAILABLE | MQGMO_WAIT
MQGET GMO.MatchOptions = MQMO_MATCH_MSG_SEQ_NUMBER, MD.MsgSeqNumber = 1
/* Examine first or only message */
 ...

GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
   MQGET
   /* Examine each remaining message in the group */
   ...
```

The above group is repeated until MQRC_NO_MSG_AVAILABLE is returned.

Browsing and retrieving destructively:

In this example, the application browses each of the logical messages within a group, before deciding whether to retrieve that group destructively.

The first part of this example is similar to the previous one. However, in this case, having browsed an entire group, we decide to go back and retrieve it destructively.

As each group is removed in this example, MQGMO_LOGICAL_ORDER is not used when looking for the first or only message in a group.

The following gives an example of browsing and then retrieving destructively:

```
GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
            | MQGMO_ALL_MESSAGES_AVAILABE | MQGMO_WAIT
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
   MQGET
   /* Examine each remaining message in the group (or as many as
      necessary to decide whether or not to get it destructively) */
   ...

if ( we want to retrieve the group destructively )

   if ( GroupStatus == ' ' )
```

```
   /* We retrieved an ungrouped message */
   GMO.Options = MQGMO_MSG_UNDER_CURSOR | MQGMO_SYNCPOINT
   MQGET GMO.MatchOptions = 0
   /* Process the message */
   ...

else
   /* We retrieved one or more messages in a group.  The browse cursor */
   /* will not normally be still on the first in the group, so we have */
   /* to match on the GroupId and MsgSeqNumber = 1.                    */
   /* Another way, which works for both grouped and ungrouped messages,*/
   /* would be to remember the MsgId of the first message when it was  */
   /* browsed, and match on that.                                      */
   GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT
   MQGET GMO.MatchOptions = MQMO_MATCH_GROUP_ID
                          | MQMO_MATCH_MSG_SEQ_NUMBER,
        (MQMD.GroupId      = value already in the MD)
         MQMD.MsgSeqNumber = 1
   /* Process first or only message */
   ...

   GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT
               | MQGMO_LOGICAL_ORDER
   do while ( GroupStatus == MQGS_MSG_IN_GROUP )
      MQGET
      /* Process each remaining message in the group */
      ...
```

Avoiding repeated delivery of browsed messages

By using certain open options and get-message options, you can mark messages as having been browsed so that they are not retrieved again by the current or other cooperating applications. Messages can be unmarked explicitly or automatically to make them available again for browsing.

If you browse messages on a queue, you might retrieve them in a different order to the order in which you would retrieve them if you got them destructively. In particular, you can browse the same message multiple times, which is not possible if it is removed from the queue. To avoid this you can *mark* messages as they are browsed, and avoid retrieving marked messages. This is sometimes referred to as *browse with mark*. To mark browsed messages, use the get message option MQGMO_MARK_BROWSE_HANDLE, and to retrieve only messages that are not marked, use MQGMO_UNMARKED_BROWSE_MSG. If you use the combination of options MQGMO_BROWSE_FIRST, MQGMO_UNMARKED_BROWSE_MSG, and MQGMO_MARK_BROWSE_HANDLE, and issue repeated MQGETs, you will retrieve each message on the queue in turn. This prevents repeated delivery of messages even though MQGMO_BROWSE_FIRST is used to ensure that messages are not skipped. This combination of options can be represented by the single constant MQGMO_BROWSE_HANDLE. When there are no messages on the queue that have not been browsed, MQRC_NO_MSG_AVAILABLE is returned.

If multiple applications are browsing the same queue, they can open the queue with the options MQOO_CO_OP and MQOO_BROWSE. The object handle returned by each MQOPEN is considered to be part of a cooperating group. Any message returned by an MQGET call specifying the option MQGMO_MARK_BROWSE_CO_OP is considered to be marked for this cooperating set of handles.

If a message has been marked for some time, it can be automatically unmarked by the queue manager and made available for browsing again. The queue manager attribute MsgMarkBrowseInterval gives the time in milliseconds for which a

message is to remain marked for the cooperating set of handles. A MsgMarkBrowseInterval of -1 means that messages are never automatically unmarked.

When the single process or set of cooperative processes marking messages stop, any marked messages become unmarked.

Examples of cooperative browsing

You might run multiple copies of a dispatcher application to browse messages on a queue and initiate a consumer based on the content of each message. In each dispatcher, open the queue with MQOO_CO_OP. This indicates that the dispatchers are cooperating and will be aware of each other's marked messages. Each dispatcher then makes repeated MQGET calls, specifying the options MQGMO_BROWSE_FIRST, MQGMO_UNMARKED_BROWSE_MSG, and MQGMO_MARK_BROWSE_CO_OP (you can use the single constant MQGMO_BROWSE_CO_OP to represent this combination of options). Each dispatcher application then retrieves only those messages that have not already been marked by other cooperating dispatchers. The dispatcher initializes a consumer and passes the MsgToken returned by the MQGET to the consumer, which destructively gets the message from the queue. If the consumer backs out the MQGET of the message, then the message is available for one of the browsers to re-dispatch, because it is no longer marked. If the consumer does not do an MQGET on the message, then after the MsgMarkBrowseInterval has passed, the queue manager unmarks the message for the cooperating set of handles, and it can be re-dispatched.

Rather than multiple copies of the same dispatcher application, you might have a number of different dispatcher applications browsing the queue, each suitable for processing a subset of the messages on the queue. In each dispatcher, open the queue with MQOO_CO_OP. This indicates that the dispatchers are cooperating and will be aware of each other's marked messages.

- If the order of message processing for a single dispatcher is important, each dispatcher makes repeated MQGET calls, specifying the options MQGMO_BROWSE_FIRST, MQGMO_UNMARKED_BROWSE_MSG , and MQGMO_MARK_BROWSE_HANDLE (or MQGMO_BROWSE_HANDLE). If the browsed message is suitable for this dispatcher to process, it then makes an MQGET call specifying MQMO_MATCH_MSG_TOKEN, MQGMO_MARK_BROWSE_CO_OP, and the MsgToken returned by the previous MQGET call. If the call succeeds, the dispatcher initializes the consumer, passing the MsgToken to it.
- If the order of message processing is not important and the dispatcher is expected to process most of the messages it encounters, use the options MQGMO_BROWSE_FIRST, MQGMO_UNMARKED_BROWSE_MSG , and MQGMO_MARK_BROWSE_CO_OP (or MQGMO_BROWSE_CO_OP). If the dispatcher browses a message it cannot process, it unmarks the message by calling MQGET with the option MQMO_MATCH_MSG_TOKEN, MQGMO_UNMARK_BROWSE_CO_OP, and the MsgToken returned previously.

Some cases where the MQGET call fails

If certain attributes of a queue are changed using the FORCE option on a command between issuing an MQOPEN and an MQGET call, the MQGET call fails and returns the MQRC_OBJECT_CHANGED reason code.

The queue manager marks the object handle as being no longer valid. This also happens if the changes apply to any queue to which the queue name resolves. The attributes that affect the handle in this way are listed in the description of the MQOPEN call in the *WebSphere MQ Application Programming Reference*. If your call returns the MQRC_OBJECT_CHANGED reason code, close the queue, reopen it, then try to get a message again.

If get operations are inhibited for a queue from which you are attempting to get messages (or any queue to which the queue name resolves), the MQGET call fails and returns the MQRC_GET_INHIBITED reason code. This happens even if you are using the MQGET call for browsing. You might be able to get a message successfully if you attempt the MQGET call at a later time, if the design of the application is such that other programs change the attributes of queues regularly.

If a dynamic queue (either temporary or permanent) has been deleted, MQGET calls using a previously-acquired object handle fail and return the MQRC_Q_DELETED reason code.

Writing data-conversion exits

Not supported in MQSeries for VSE/ESA™.

When you do an MQPUT, your application creates the message descriptor (MQMD) of the message. Because WebSphere MQ needs to be able to understand the contents of the MQMD regardless of the platform it is created on, it is converted automatically by the system.

Application data, however, is not converted automatically. If character data is being exchanged between platforms where the *CodedCharSetId* and *Encoding* fields differ, for example, between ASCII and EBCDIC, the application must arrange for conversion of the message. Application data conversion can be performed by the queue manager itself or by a user exit program, referred to as a *data-conversion exit*. The queue manager can perform data conversion itself, using one of its built-in conversion routines, if the application data is in one of the built-in formats (such as MQFMT_STRING). This chapter discusses the data-conversion exit facility that WebSphere MQ provides for when the application data is not in a built-in format.

Control can be passed to the data-conversion exit during an MQGET call. This avoids converting across different platforms before reaching the final destination. However, if the final destination is a platform that does not support data conversion on the MQGET, you must specify CONVERT(YES) on the sender channel that sends the data to its final destination. This ensures that WebSphere MQ converts the data during transmission. In this case, your data-conversion exit must reside on the system where the sender channel is defined.

The MQGET call is issued directly by the application. Set the *CodedCharSetId* and *Encoding* fields in the MQMD to the character set and encoding required. If your application uses the same character set and encoding as the queue manager, set *CodedCharSetId* to MQCCSI_Q_MGR, and *Encoding* to MQENC_NATIVE. After the MQGET call completes, these fields have the values appropriate to the message data returned. These might differ from the values required if the conversion was not successful. Your application should reset these fields to the values required before each MQGET call.

The conditions required for the data-conversion exit to be called are defined for the MQGET call in the *WebSphere MQ Application Programming Reference*.

For a description of the parameters that are passed to the data-conversion exit, and detailed usage notes, see the *WebSphere MQ Application Programming Reference* for the MQ_DATA_CONV_EXIT call and the MQDXP structure.

Programs that convert application data between different machine encodings and CCSIDs must conform to the WebSphere MQ data conversion interface (DCI).

This chapter introduces data-conversion exits, under these headings:

Invoking the data-conversion exit

A data-conversion exit is a user-written exit that receives control during the processing of an MQGET call.

The exit is invoked if the following are true:

- The MQGMO_CONVERT option is specified on the MQGET call.
- Some or all of the message data is not in the requested character set or encoding.
- The *Format* field in the MQMD structure associated with the message is not MQFMT_NONE.
- The *BufferLength* specified on the MQGET call is not zero.
- The message data length is not zero.
- The message contains data that has a user-defined format. The user-defined format can occupy the entire message, or be preceded by one or more built-in formats. For example, the user-defined format might be preceded by an MQFMT_DEAD_LETTER_HEADER format. The exit is invoked to convert only the user-defined format; the queue manager converts any built-in formats that precede the user-defined format.

  A user-written exit can also be invoked to convert a built-in format, but this happens only if the built-in conversion routines cannot convert the built-in format successfully.

There are some other conditions, described fully in the usage notes of the MQ_DATA_CONV_EXIT call in the *WebSphere MQ Application Programming Reference*.

See the *WebSphere MQ Application Programming Reference* for details of the MQGET call. Data-conversion exits cannot use MQI calls, other than MQXCNVC.

A new copy of the exit is loaded when an application attempts to retrieve the first message that uses that *Format* since the application connected to the queue manager. A new copy might also be loaded at other times if the queue manager has discarded a previously-loaded copy.

The data-conversion exit runs in an environment similar to that of the program that issued the MQGET call. As well as user applications, the program can be an MCA (message channel agent) sending messages to a destination queue manager that does not support message conversion. The environment includes address space and user profile, where applicable. The exit cannot compromise the queue manager's integrity, because it does not run in the queue manager's environment.

In a client-server environment, the exit is loaded at the server, and conversion takes place there.

Data conversion on z/OS

On z/OS, be aware of the following:

- Exit programs can be written in assembler language only.
- Exit programs must be reentrant, and capable of running anywhere in storage.
- Exit programs must restore the environment on exit to that at entry, and must free any storage obtained.
- Exit programs must not WAIT, or issue ESTAEs or SPIEs.
- Exit programs are usually invoked as if by z/OS LINK in:
  - Non-authorized problem program state
  - Primary address space control mode
  - Non cross-memory mode
  - Non access-register mode
  - 31 bit addressing mode
  - TCB-PRB mode
- When used by a CICS application, the exit is invoked by EXEC CICS LINK, and must conform to the CICS programming conventions. The parameters are passed by pointers (addresses) in the CICS communication area (COMMAREA).

  Although not recommended, user exit programs can also use CICS API calls, with the following caution:
  - Do not issue syncpoints, as the results could influence units of work declared by the MCA.
  - Do not update any resources controlled by a resource manager other than WebSphere MQ for z/OS, including those controlled by CICS Transaction Server for OS/390.
- For distributed queuing without CICS, the exit is loaded from the data set referenced by the CSQXLIB DD statement.
- For distributed queuing using CICS, data-conversion exits are not supported.

Writing a data-conversion exit program

For z/OS, you must write data-conversion exits in assembler language. For other platforms, it is recommended that you use the C programming language.

To help you to create a data-conversion exit program, the following are supplied:

- A skeleton source file
- A convert characters call

- A utility that creates a fragment of code that performs data conversion on data type structures This utility takes C input only. On z/OS, it produces assembler code.

These are described in subsequent sections.

For the procedure for writing the programs see:

- "Writing a data-conversion exit program for WebSphere MQ for i5/OS" on page 170
- "Writing a data-conversion exit program for WebSphere MQ for z/OS" on page 171
- "Writing a data-conversion exit for WebSphere MQ on UNIX systems" on page 172
- "Writing a data-conversion exit for WebSphere MQ for Windows" on page 178

Skeleton source file

These can be used as your starting point when writing a data-conversion exit program.

The files supplied are listed in Table 10.

Table 10. Skeleton source files

Platform	File
AIX	amqsvfc0.c
i5/OS	QMQMSAMP/QCSRC(AMQSVFC4)
HP-UX	amqsvfc0.c
Linux	amqsvfc0.c
z/OS	CSQ4BAX8 (1) CSQ4BAX9 (2) CSQ4CAX9 (3)
Solaris	amqsvfc0.c
Windows systems	amqsvfc0.c

Notes:

1. Illustrates the MQXCVNC call.
2. A wrapper for the code fragments generated by the utility for use in all environments except CICS.
3. A wrapper for the code fragments generated by the utility for use in the CICS environment.

Convert characters call

Use the MQXCNVC (convert characters) call from within a data-conversion exit program to convert character message data from one character set to another. For certain multibyte character sets (for example, UCS2 character sets), the appropriate options must be used.

No other MQI calls can be made from within the exit; an attempt to make such a call fails with reason code MQRC_CALL_IN_PROGRESS.

See the *WebSphere MQ Application Programming Reference* for further information on the MQXCNVC call and appropriate options.

Utility for creating conversion-exit code

The commands for creating conversion-exit code are:

i5/OS CVTMQMDTA (Convert WebSphere MQ Data Type)

Windows systems and UNIX systems
`crtmqcvx` (Create WebSphere MQ conversion-exit)

z/OS CSQUCVX

The command for your platform produces a fragment of code that performs data conversion on data type structures, for use in your data-conversion exit program. The command takes a file containing one or more C language structure definitions. On z/OS, it then generates a data set containing assembler code fragments and conversion functions. On other platforms, it generates a file with a C function to convert each structure definition. On z/OS, the utility requires access to the LE/370 run-time library SCEERUN.

Invoking the CSQUCVX utility on z/OS:

Figure 17 shows an example of the JCL used to invoke the CSQUCVX utility.

```
//CVX    EXEC PGM=CSQUCVX
//STEPLIB  DD DISP=SHR,DSN=thlqual.SCSQANLE
//         DD DISP=SHR,DSN=thlqual.SCSQLOAD
//         DD DISP=SHR,DSN=le370qual.SCEERUN
//SYSPRINT DD SYSOUT=*
//CSQUINP  DD DISP=SHR,DSN=MY.MQSERIES.FORMATS(MSG1)
//CSQUOUT  DD DISP=OLD,DSN=MY.MQSERIES.EXITS(MSG1)
```

Figure 17. Sample JCL used to invoke the CSQUCVX utility

z/OS data definition statements:

The CSQUCVX utility requires DD statements with the following DDnames:

SYSPRINT
: This specifies a data set or print spool class for reports and error messages.

CSQUINP
: This specifies the sequential data set containing the definitions of the data structures to be converted.

CSQUOUT
: This specifies the sequential data set where the conversion code fragments are to be written. The logical record length (LRECL) must be 80 and the record format (RECFM) must be FB.

Error messages in Windows systems, and UNIX systems:

The `crtmqcvx` command returns messages in the range AMQ7953 through AMQ7970.

These are listed in *WebSphere MQ Messages*.

There are two main types of error:

- Major errors, such as syntax errors, when processing cannot continue.

  A message is displayed on the screen giving the line number of the error in the input file. The output file might have been partially created.
- Other errors when a message is displayed stating that a problem has been found but that parsing of the structure can continue.

  The output file has been created and contains error information on the problems that have occurred. This error information is prefixed by `#error` so that the code produced is not accepted by any compiler without intervention to rectify the problems.

Valid syntax

Your input file for the utility must conform to the C language syntax.

If you are unfamiliar with C, refer to "Example of valid syntax for the input data set" on page 169.

In addition, be aware of the following rules:

- typedef is recognized only before the struct keyword.
- A structure tag is required on your structure declarations.
- You can use empty square brackets [ ] to denote a variable length array or string at the end of a message.
- Multidimensional arrays and arrays of strings are not supported.
- The following additional data types are recognized:
  - MQBOOL
  - MQBYTE
  - MQCHAR
  - MQFLOAT32
  - MQFLOAT64
  - MQSHORT
  - MQLONG
  - MQINT8
  - MQUINT8
  - MQINT16
  - MQUINT16
  - MQINT32
  - MQUINT32
  - MQINT64
  - MQUINT64

  MQCHAR fields are code page converted, but MQBYTE, MQINT8 and MQUINT8 are left untouched. If the encoding is different, MQSHORT, MQLONG, MQINT16, MQUINT16, MQINT32, MQUINT32, MQINT64, MQUINT64, MQFLOAT32, MQFLOAT64 and MQBOOL are converted accordingly.
- Do *not* use the following:

– double
– pointers
– bit-fields

This is because the utility for creating conversion-exit code does not provide the facility to convert these data types. To overcome this, you can write your own routines and call them from the exit.

Other points to note:

- Do not use sequence numbers in the input data set.
- If there are fields for which you want to provide your own conversion routines, declare them as MQBYTE, and then replace the generated CMQXCFBA macros with your own conversion code.

Example of valid syntax for the input data set:

```
struct TEST { MQLONG    SERIAL_NUMBER;
              MQCHAR    ID[5];
              MQINT16   VERSION;
              MQBYTE    CODE[4];
              MQLONG    DIMENSIONS[3];
              MQCHAR    NAME[24];
            } ;
```

This corresponds to the following declarations in other programming languages:

COBOL:

```
       10 TEST.
        15 SERIAL-NUMBER  PIC S9(9) BINARY.
        15 ID             PIC X(5).
        15 VERSION        PIC S9(4) BINARY.
      * CODE IS NOT TO BE CONVERTED
        15 CODE           PIC X(4).
        15 DIMENSIONS     PIC S9(9) BINARY OCCURS 3 TIMES.
        15 NAME           PIC X(24).
```

System/390 assembler:

Supported only on WebSphere MQ for z/OS.

```
TEST           EQU *
SERIAL_NUMBER  DS  F
ID             DS  CL5
VERSION        DS  H
CODE           DS  XL4
DIMENSIONS     DS  3F
NAME           DS  CL24
```

PL/I:

Supported on z/OS only

```
 DCL 1 TEST,
       2 SERIAL_NUMBER  FIXED BIN(31),
       2 ID             CHAR(5),
       2 VERSION        FIXED BIN(15),
       2 CODE           CHAR(4),        /* not to be converted */
       2 DIMENSIONS(3)  FIXED BIN(31),
       2 NAME           CHAR(24);
```

Writing a data-conversion exit program for WebSphere MQ for i5/OS

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD. The *Format* name must not have leading embedded blanks, and trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long. Remember to use this name each time that you send a message (our example uses the name Format).
2. Create a structure to represent your message. See "Valid syntax" on page 168 for an example.
3. Run this structure through the CVTMQMDTA command to create a code fragment for your data-conversion exit.

   The functions generated by the CVTMQMDTA command use macros that are shipped in the file QMQM/H(AMQSVMHA). These macros are written assuming that all structures are packed; amended them if this is not the case.
4. Take a copy of the supplied skeleton source file, QMQMSAMP/QCSRC(AMQSVFC4) and rename it. (Our example uses the name EXIT_MOD.)
5. Find the following comment boxes in the source file and insert code as described:
   a. Toward the bottom of the source file, a comment box starts with:

      ```
      /* Insert the functions produced by the data-conversion exit */
      ```

      Here, insert the code fragment generated in step 3.
   b. Near the middle of the source file, a comment box starts with:

      ```
      /* Insert calls to the code fragments to convert the format's */
      ```

      This is followed by a commented-out call to the function `ConverttagSTRUCT`.

      Change the name of the function to the name of the function that you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.
   c. Near the top of the source file, a comment box starts with:

      ```
      /* Insert the function prototypes for the functions produced by */
      ```

      Here, insert the function prototype statements for the functions added in step 5a above.

   If the message contains character data, the generated code calls MQXCNVC; this can be resolved by binding the service program QMQM/LIBMQM.
6. Compile the source module, EXIT_MOD, as follows:

   ```
   CRTCMOD MODULE(library/EXIT_MOD) +
   SRCFILE(QCSRC) +
   TERASPACE(*YES *TSIFC)
   ```
7. Create/link the program.

   For nonthreaded applications, use the following:

   ```
   CRTPGM PGM(library/Format)  +
      MODULE(library/EXIT_MOD)  +
      BNDSRVPGM(QMQM/LIBMQM)  +
      ACTGRP(QMQM)  +
      USRPRF(*USER)
   ```

   In addition to creating the data-conversion exit for the basic environment, another is required in the threaded environment. This loadable object must be

followed by _R. Use the LIBMQM_R library to resolve calls to the MQXCNVC. Both loadable objects are required for a threaded environment.

```
CRTPGM PGM(library/Format_R)  +
   MODULE(library/EXIT_MOD)  +
   BNDSRVPGM(QMQM/LIBMQM_R)  +
   ACTGRP(QMQM)  +
   USRPRF(*USER)
```

8. Place the output in the library list for the WebSphere MQ job. It is recommended that, for production, data-conversion exit programs be stored in QSYS.

Note:

1. If CVTMQMDTA uses packed structures, all WebSphere MQ applications must use the _Packed qualifier.
2. Data-conversion exit programs must be reentrant.
3. MQXCNVC is the *only* MQI call that can be issued from a data-conversion exit.
4. Compile the exit program with the user profile compiler option set to *USER, so that the exit runs with the authority of the user.
5. Teraspace memory enablement is required for all user exits with WebSphere MQ for i5/OS; specify TERASPACE(*YES *TSIFC) in the CRTCMOD and CRTBNDC commands.

Writing a data-conversion exit program for WebSphere MQ for z/OS

Follow these steps:

1. Take the supplied source skeleton CSQ4BAX9 (for non-CICS environments) or CSQ4CAX9 (for CICS) as your starting point.
2. Run the CSQUCVX utility.
3. Follow the instructions in the prolog of CSQ4BAX9 or CSQ4CAX9 to incorporate the routines generated by the CSQUCVX utility, in the order that the structures occur in the message that you want to convert.
4. The utility assumes that the data structures are not packed, that the implied alignment of the data is honored, and that the structures start on a fullword boundary, with bytes being skipped as required (as between ID and VERSION in the “Example of valid syntax for the input data set” on page 169). If the structures are packed, omit the CMQXCALA macros that are generated. You are therefore strongly recommended to declare your structures in such a way that all fields are named and no bytes are skipped; in the “Example of valid syntax for the input data set” on page 169, add a field “MQBYTE DUMMY;” between ID and VERSION.
5. The supplied exit returns an error if the input buffer is shorter than the message format to be converted. Although the exit converts as many complete fields as possible, the error causes an unconverted message to be returned to the application. If you want to allow short input buffers to be converted as far as possible, including partial fields, change the TRUNC= value on the CSQXCDFA macro to YES: no error is returned, so the application receives a converted message. The application must handle the truncation.
6. Add any other special processing code that you need.
7. Rename the program to your data format name.

8. Compile and link-edit your program like a batch application program (unless it is for use with CICS applications). The macros in the code generated by the utility are in the library, **thlqual**.SCSQMACS.

   If the message contains character data, the generated code calls MQXCNVC. If your exit uses this call, link-edit it with the exit stub program CSQASTUB. The stub is language-independent and environment-independent. Alternatively, you can load the stub dynamically using the dynamic call name CSQXCNVC. See "Dynamically calling the WebSphere MQ stub" on page 377 for more information.

   Place the link-edited module in your application load library, and in a data set that is referenced by the CSQXLIB DD statement of your task procedure started by your channel initiator.
9. If the exit is for use by CICS applications, compile and link-edit it like a CICS application program, including CSQASTUB if required. Place it in your CICS application program library. Define the program to CICS in the usual way, specifying EXECKEY(CICS) in the definition.

Note: Although the LE/370 run-time libraries are needed for running the CSQUCVX utility (see step 2 on page 171), they are not needed for link-editing or running the data-conversion exit itself (see steps 8 and 9).

See "Writing WebSphere MQ-IMS bridge applications" on page 327 for information about data conversion within the WebSphere MQ-IMS bridge.

Writing a data-conversion exit for WebSphere MQ on UNIX systems

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD, and be in uppercase, for example, MYFORMAT. The *Format* name must not have leading blanks. Trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long. Remember to use this name each time that you send a message.
2. Create a structure to represent your message. See "Valid syntax" on page 168 for an example.
3. Run this structure through the `crtmqcvx` command to create a code fragment for your data-conversion exit.

   The functions generated by the `crtmqcvx` command use macros that assume that all structures are packed; amend them if this is not the case.
4. Copy the supplied skeleton source file, renaming it to the name of your message format that you set in step 1. The skeleton source file, and the copy, are read-only.

   The skeleton source file is called amqsvfc0.c.
5. On WebSphere MQ for AIX, a skeleton export file called amqsvfc.exp is also supplied. Copy this file, renaming it to MYFORMAT.EXP.
6. The skeleton includes a sample header file, amqsvmha.h, in the directory /usr/mqm/inc (on AIX) or /opt/mqm/inc (on other UNIX systems). Make sure that your include path points to this directory to pick up this file.

   The amqsvmha.h file contains macros that are used by the code generated by the `crtmqcvx` command. If the structure to be converted contains character data, these macros call MQXCNVC.

7. Find the following comment boxes in the source file and insert code as described:
   a. Toward the bottom of the source file, a comment box starts with:
      ```
      /* Insert the functions produced by the data-conversion exit */
      ```
      Here, insert the code fragment generated in step 3 on page 172.
   b. Near the middle of the source file, a comment box starts with:
      ```
      /* Insert calls to the code fragments to convert the format's */
      ```
      This is followed by a commented-out call to the function `ConverttagSTRUCT`.
      Change the name of the function to the name of the function that you added in step 7a. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.
   c. Near the top of the source file, a comment box starts with:
      ```
      /* Insert the function prototypes for the functions produced by */
      ```
      Here, insert the function prototype statements for the functions added in step 3 on page 172 above.
8. Resolve this call by linking the routine with the library libmqm. For threaded programs, the routine must be linked with the library libmqm_r (AIX and HP-UX only).
9. Compile your exit as a shared library, using MQStart as the entry point. To do this, see "Compiling data-conversion exits on UNIX and Linux" on page 174.
10. Place the output in the default system directory, /var/mqm/exits, to ensure that it can be loaded when required. The path used to look for the data-conversion exits is given in the qm.ini file. This path can be set for each queue manager and the exit is only looked for in that path or paths.

Note:

1. If `crtmqcvx` uses packed structures, all WebSphere MQ applications must be compiled in this way.
2. Data-conversion exit programs must be reentrant.
3. MQXCNVC is the *only* MQI call that can be issued from a data-conversion exit.

UNIX environment

You need to take into consideration if you are building 32–bit or 64–bit applications and whether or not you are in a threaded or non threaded environment.

Non-threaded environment:

The loadable object must have its name in upper case, for example MYFORMAT. Use the libmqm library to resolve the calls to MQXCNVC.

Threaded environment:

In addition to creating the data-conversion exit for the basic environment, another is required in the threaded environment.

This loadable object must be followed by _r (on AIX, HP-UX, and Linux) to indicate that it is a threaded version. Use the libmqm_r library to resolve the calls to MQXCNVC. Note that both loadable objects (non-threaded and threaded) are required for a threading environment.

If you are running MQI clients, all data conversion is performed by the proxy running on the machine to which the client is attached. This means that any data conversion exits are run on the server, in the environment of the proxy, and not as part of the client application.

For most platforms, the proxy/responder program is a threaded program. Consequently, the data conversion exit must be compiled with appropriate options to run in this threaded environment. Whether or not the client application is threaded is irrelevant.

On the WebSphere MQ for UNIX systems, the proxy is threaded.

Note: If the data-conversion exits are in a mixed non-threaded and threaded environment, the calling environment is detected and the appropriate object loaded. The shared object should be placed in /var/mqm/exits, or var/mqm/exits64 to ensure it can be loaded when required.

Compiling data-conversion exits on UNIX and Linux

The following sections give examples of how to compile a data conversion exit on UNIX platforms and Linux.

On all platforms, the entry point to the module is MQStart.

On AIX:

32 bit applications:

Non-threaded:

Compile the exit source code by issuing the following command:

```
cc -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits/MYFORMAT \
    MYFORMAT.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqmzf
```

Threaded:

Compile the exit source code by issuing the following command:

```
xlc_r -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits/MYFORMAT_r \
    MYFORMAT.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqmzf_r
```

64 bit applications:

Non-threaded:

Compile the exit source code by issuing the following command:

```
cc -q64 -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits64/MYFORMAT \
    MYFORMAT.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqmzf
```

Threaded:

Compile the API exit source code by issuing the following command:

```
xlc_r -q64 -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits64/MYFORMAT_r \
    MYFORMAT.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqmzf_r
```

On HP-UX:

PA-RISC platform:

32 bit applications:

Non-threaded:

1. Compile the exit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld +b: -b MYFORMAT.o +ee MQStart -o  \
    /var/mqm/exits/MYFORMAT -L/opt/mqm/lib -L/usr/lib -lmqmzf
rm MYFORMAT.o
```

Threaded:

1. Compile the exit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld +b: -b MYFORMAT.o +ee MQStart -o  \
   /var/mqm/exits/MYFORMAT_r -L/opt/mqm/lib -L/usr/lib -lmqmzf_r -lpthread
rm MYFORMAT.o
```

64 bit applications:

Non-threaded:

1. Compile the exit source code

```
 c89 +DD64 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld -b +noenvvar MYFORMAT.o +ee MQStart \
    -o  /var/mqm/exits64/MYFORMAT -L/opt/mqm/lib64 \
    -L/usr/lib/pa20_64 -lmqmzf
rm MYFORMAT.o
```

Threaded:

1. Compile the exit source code

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld -b +noenvvar MYFORMAT.o +ee MQStart  \
  -o  /var/mqm/exits64/MYFORMAT_r -L/opt/mqm/lib64   \
  -L/usr/lib/pa20_64 -lmqmzf_r -lpthread
rm MYFORMAT.o
```

Itanium platform:

32 bit applications:

Non-threaded:

1. Compile the exit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld +b: -b MYFORMAT.o +ee MQStart -o  \
    /var/mqm/exits/MYFORMAT -L/opt/mqm/lib -L/usr/lib/hpux32 -lmqmzf
rm MYFORMAT.o
```

Threaded:

1. Compile the exit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld +b: -b MYFORMAT.o +ee MQStart -o  \
    /var/mqm/exits/MYFORMAT_r -L/opt/mqm/lib -L/usr/lib/hpux32 \
    -lmqmzf_r -lpthread
rm MYFORMAT.o
```

64 bit applications:

Non-threaded:

1. Compile the exit source code

```
 c89 +DD64 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld -b +noenvvar MYFORMAT.o +ee MQStart \
    -o  /var/mqm/exits64/MYFORMAT -L/opt/mqm/lib64 \
    -L/usr/lib/hpux64 -lmqmzf
rm MYFORMAT.o
```

Threaded:

1. Compile the exit source code

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o MYFORMAT.o MYFORMAT.c -I/opt/mqm/inc
```

2. Link the exit object

```
ld -b +noenvvar MYFORMAT.o +ee MQStart  \
    -o  /var/mqm/exits64/MYFORMAT_r -L/opt/mqm/lib64   \
    -L/usr/lib/hpux64 -lmqmzf_r -lpthread
rm MYFORMAT.o
```

On Linux:

31 bit applications (zSeries platform):

Non-threaded:

Compile the exit source code by issuing the following command:

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/MYFORMAT MYFORMAT.c \
     -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib \
     -Wl,-rpath=/usr/lib -lmqmzf
```

Threaded:

Compile the exit source code by issuing the following command:

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/MYFORMAT_r MYFORMAT.c
     -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib
     -Wl,-rpath=/usr/lib -lmqmzf_r
```

32 bit applications:

Non-threaded:

Compile the exit source code by issuing the following command:

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/MYFORMAT MYFORMAT.c
     -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib
     -Wl,-rpath=/usr/lib -lmqmzf
```

Threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/MYFORMAT_r MYFORMAT.c
     -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib
     -Wl,-rpath=/usr/lib -lmqmzf_r
```

64 bit applications:

Non-threaded:

Compile the exit source code by issuing the following command:

```
 gcc -m64 -shared -fPIC -o /var/mqm/exits64/MYFORMAT MYFORMAT.c
      -I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
      -Wl,-rpath=/usr/lib64 -lmqmzf
```

Threaded:

Compile the exit source code by issuing the following command:

```
 gcc -m64 -shared -fPIC -o /var/mqm/exits64/MYFORMAT_r MYFORMAT.c
      -I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
      -Wl,-rpath=/usr/lib64 -lmqmzf_r
```

On Solaris:

SPARC platform:

32 bit applications:

Compile the exit source code by issuing the following command:

```
cc -xarch=v8plus -KPIC -mt -G -o /var/mqm/exits/MYFORMAT \
   MYFORMAT.c -I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib \
   -R/usr/lib/32 -lmqm  -lmqmcs  -lmqmzse -lsocket -lnsl -ldl
```

64 bit applications:

Compile the API exit source code by issuing the following command:

```
cc -xarch=v9 -KPIC -mt -G -o /var/mqm/exits64/MYFORMAT \
   MYFORMAT.c -I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 \
   -R/usr/lib/64  -lmqm  -lmqmcs  -lmqmzse -lsocket -lnsl -ldl
```

x86-64 platform:

32 bit applications:

Compile the API exit source code by issuing the following command:

```
cc -xarch=386 -KPIC -mt -G -o /var/mqm/exits/MYFORMAT \
   MYFORMAT.c -I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib \
   -R/usr/lib/32 -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket \
   -lnsl -ldl
```

64 bit applications:

Compile the exit source code by issuing the following command:

```
cc -xarch=amd64 -KPIC -mt -G -o /var/mqm/exits64/MYFORMAT \
   MYFORMAT.c -I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 \
   -R/usr/lib/64  -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket \
   -lnsl -ldl
```

Writing a data-conversion exit for WebSphere MQ for Windows

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD. The *Format* name must not have leading blanks. Trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long.

   A .DEF file called amqsvfcn.def is also supplied in the samples directory, `<drive:\directory>\Program Files\IBM\WebSphere MQ\Tools\C\Samples`. Take a copy of this file and rename it, for example, to MYFORMAT.DEF. Make sure that the name of the DLL being created and the name specified in MYFORMAT.DEF are the same. Overwrite the name FORMAT1 in MYFORMAT.DEF with the new format name.

   Remember to use this name each time that you send a message.
2. Create a structure to represent your message. See "Valid syntax" on page 168 for an example.
3. Run this structure through the `crtmqcvx` command to create a code fragment for your data-conversion exit.

   The functions generated by the CRTMQCVX command use macros that are written assuming that all structures are packed; amend them if this is not the case.
4. Copy the supplied skeleton source file, amqsvfc0.c, renaming it to the name of your message format that you set in step 1.

   amqsvfc0.c is in `<drive:\directory>\Tools\C\Samples` where `<drive:\directory>` is the directory specified on installation. (The default installation directory is `C:\Program Files\IBM\WebSphere MQ`.)

   The skeleton includes a sample header file amqsvmha.h in the same directory. Make sure that your include path points to this directory to pick up this file.

   The amqsvmha.h file contains macros that are used by the code generated by the CRTMQCVX command. If the structure to be converted contains character data, these macros call MQXCNVC.
5. Find the following comment boxes in the source file and insert code as described:

   a. Toward the bottom of the source file, a comment box starts with:

      ```
      /* Insert the functions produced by the data-conversion exit */
      ```

      Here, insert the code fragment generated in step 3.

   b. Near the middle of the source file, a comment box starts with:

      ```
      /* Insert calls to the code fragments to convert the format's */
      ```

      This is followed by a commented-out call to the function `ConverttagSTRUCT`.

      Change the name of the function to the name of the function that you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

   c. Near the top of the source file, a comment box starts with:

      ```
      /* Insert the function prototypes for the functions produced by */
      ```

      Here, insert the function prototype statements for the functions added in step 3 above.

6. Resolve this call by linking the routine with the library MQMVX.LIB, in the directory <drive:\directory>\Program Files\IBM\WebSphere MQ\Tools\Lib. The 64-bit version of this library is in the directory: <drive:\directory>\Program Files\IBM\WebSphere MQ\Tools\Lib64
7. For 32-bit user exits create the following command file:

```
cl -I <drive:\dir>\Program Files\IBM\WebSphere MQ\Tools\C\Include -Tp   \
MYFORMAT.C -LD -DEFAULTLIB \
<drive:\dir>\Program Files\IBM\WebSphere MQ\Tools\Lib\mqm.lib  \
<drive:\dir>\Program Files\IBM\WebSphere MQ\Tools\Lib\mqmvx.lib   \
MYFORMAT.DEF
```

   For 64-bit user exits, create the following command file:

```
cl -I <drive:\dir>\Program Files (x86)\IBM\WebSphere MQ\Tools\C\Include -Tp   \
MYFORMAT.C -LD -DEFAULTLIB \
<drive:\dir>\Program Files (x86)\IBM\WebSphere MQ\Tools\Lib64\mqm.lib  \
<drive:\dir>\Program Files (x86)\IBM\WebSphere MQ\Tools\Lib64\mqmvx.lib   \
MYFORMAT.DEF
```

   where <drive:\dir> is specified at installation,

   Issue the command file to compile your exit as a DLL file.
8. Place the output in the *User_Exits* subdirectory below the WebSphere MQ data directory:

   32-bit systems
   Exits

   64-bit systems
   Exits64

   Unless you have modified the ExitsDefaultPath, the default directory for installing your exits is:

   C:\Program Files\IBM\WebSphere MQ*User_Exits*

   The path used to look for the data-conversion exits is given in the registry. The registry folder is:

   HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Configuration\ClientExitPath\

   and the registry key is: ExitsDefaultPath. This path can be set for each queue manager and the exit is only looked for in that path or paths.

Note:

1. If CRTMQCVX uses packed structures, all WebSphere MQ applications must be compiled in this way.
2. Data-conversion exit programs must be reentrant.
3. MQXCNVC is the *only* MQI call that can be issued from a data-conversion exit.

Exit and switch load files on Windows operating systems

The WebSphere MQ for Windows, Version 7.0 queue manager processes are 32-bit, as a result when using 64-bit applications, some types of exit and XA switch load files will also need to have a 32-bit version available for use by the queue manager. If the 32-bit version of the exit or XA switch load file is required and is not available, then the relevant MQ API or command will fail.

Two attributes are supported in the mqs.ini and qm.ini files for ClientExitPath. These are ExitsDefaultPath=install_location\exits and ExitsDefaultPath64=install_location\exits64. Using these ensures that the appropriate library can be found. If an exit is used in a WebSphere MQ cluster, this will also ensure that the appropriate library on a remote system will be found.

The following table lists the different types of Exit and Switch load files and notes whether 32-bit or 64-bit versions, or both, are required, according to whether 32-bit or 64-bit applications are being used:

File types	32-bit applications	64-bit applications
API-crossing exit	32-bit	32-bit and 64-bit
Data conversion exit	32-bit	64-bit
Server Channel exits (all types)	32-bit	32-bit
Client Channel exits (all types)	32-bit	64-bit
Installable service exit	32-bit	32-bit
Service trace module	32-bit	32-bit and 64-bit
Cluster WLM exit	32-bit	32-bit
Pub/Sub routing exit	32-bit	32-bit
Database switch load files	32-bit	32-bit and 64-bit
External Transaction Manager AX libraries	32-bit	64-bit

Inquiring about and setting object attributes

Attributes are the properties that define the characteristics of a WebSphere MQ object.

They affect the way that a queue manager processes an object. The attributes of each type of WebSphere MQ object are described in detail in the *WebSphere MQ Application Programming Reference*.

Some attributes are set when the object is defined, and can be changed only by using the WebSphere MQ commands; an example of such an attribute is the default priority for messages put on a queue. Other attributes are affected by the operation of the queue manager and can change over time; an example is the current depth of a queue.

You can inquire about the current values of most attributes using the MQINQ call. The MQI also provides an MQSET call with which you can change some queue attributes. You cannot use the MQI calls to change the attributes of any other type of object; instead you must use:

For WebSphere MQ for z/OS
The ALTER operator commands (or the DEFINE commands with the REPLACE option), which are described in the *WebSphere MQ Script (MQSC) Command Reference*.

For WebSphere MQ for i5/OS
The CHGMQMx CL commands, which are described in the *WebSphere MQ for i5/OS System Administration Guide*, or you can use the MQSC facility.

For WebSphere MQ for all other platforms
The MQSC facility, described in the *WebSphere MQ Script (MQSC) Command Reference*.

Note: The names of the attributes of objects are shown in this book in the form that you use them with the MQINQ and MQSET calls. When you use WebSphere

MQ commands to define, alter, or display the attributes, you must identify the attributes using the keywords shown in the descriptions of the commands in the above books.

Both the MQINQ and the MQSET calls use arrays of selectors to identify those attributes that you want to inquire about or set. There is a selector for each attribute that you can work with. The selector name has a prefix, determined by the nature of the attribute:

MQCA_	These selectors refer to attributes that contain character data (for example, the name of a queue).
MQIA_	These selectors refer to attributes that contain either numeric values (such as *CurrentQueueDepth*, the number of messages on a queue) or a constant value (such as *SyncPoint*, whether or not the queue manager supports syncpoints).

Before you use the MQINQ or MQSET calls your application must be connected to the queue manager, and you must use the MQOPEN call to open the object for setting or inquiring about attributes. These operations are described in "Connecting to and disconnecting from a queue manager" on page 91 and "Opening and closing objects" on page 99.

Inquiring about the attributes of an object

Use the MQINQ call to inquire about the attributes of any type of WebSphere MQ object.

As input to this call, you must supply:

- A connection handle.
- An object handle.
- The number of selectors.
- An array of attribute selectors, each selector having the form MQCA_* or MQIA_*. Each selector represents an attribute whose value you want to inquire about, and each selector must be valid for the type of object that the object handle represents. You can specify selectors in any order.
- The number of integer attributes that you are inquiring about. Specify zero if you are not inquiring about integer attributes.
- The length of the character attributes buffer in *CharAttrLength*. This must be at least the sum of the lengths required to hold each character attribute string. Specify zero if you are not inquiring about character attributes.

The output from MQINQ is:

- A set of integer attribute values copied into the array. The number of values is determined by *IntAttrCount*. If either *IntAttrCount* or *SelectorCount* is zero, this parameter is not used.
- The buffer in which character attributes are returned. The length of the buffer is given by the *CharAttrLength* parameter. If either *CharAttrLength* or *SelectorCount* is zero, this parameter is not used.
- A completion code. If the completion code gives a warning, this means that the call completed only partially. In this case, examine the reason code.
- A reason code. There are three partial-completion situations:
  - The selector does not apply to the queue type
  - There is not enough space allowed for integer attributes
  - There is not enough space allowed for character attributes

If more than one of these situations arise, the first one that applies is returned.

If you open a queue for output or inquire and it resolves to a non-local cluster queue you can only inquire the queue name, queue type, and common attributes. The values of the common attributes are those of the chosen queue if MQOO_BIND_ON_OPEN was used. The values are those of an arbitrary one of the possible cluster queues if either MQOO_BIND_NOT_FIXED was used or MQOO_BIND_AS_Q_DEF was used and the *DefBind* queue attribute was MQBND_BIND_NOT_FIXED. See *WebSphere MQ Queue Manager Clusters* for more information.

Note: The values returned by the call are a snapshot of the selected attributes. The attributes can change before your program acts on the returned values.

There is a description of the MQINQ call in the *WebSphere MQ Application Programming Reference*.

Some cases where the MQINQ call fails

If you open an alias to inquire about its attributes, you are returned the attributes of the alias queue (the WebSphere MQ object used to access another queue), not those of the base queue.

However, the definition of the base queue to which the alias resolves is also opened by the queue manager, and if another program changes the usage of the base queue in the interval between your MQOPEN and MQINQ calls, your MQINQ call fails and returns the MQRC_OBJECT_CHANGED reason code. The call also fails if the attributes of the alias queue object are changed.

Similarly, when you open a remote queue to inquire about its attributes, you are returned the attributes of the local definition of the remote queue only.

If you specify one or more selectors that are not valid for the type of queue about whose attributes you are inquiring, the MQINQ call completes with a warning and sets the output as follows:

- For integer attributes, the corresponding elements of *IntAttrs* are set to MQIAV_NOT_APPLICABLE.
- For character attributes, the corresponding portions of the *CharAttrs* string are set to asterisks.

If you specify one or more selectors that are not valid for the type of object about whose attributes you are inquiring, the MQINQ call fails and returns the MQRC_SELECTOR_ERROR reason code.

You cannot call MQINQ to look at a model queue; use either the MQSC facility or the commands available on your platform.

Setting queue attributes

You can set only the following queue attributes using the MQSET call:

- *InhibitGet* (but not for remote queues)
- *DistList* (not on z/OS)
- *InhibitPut*
- *TriggerControl*
- *TriggerType*
- *TriggerDepth*

- *TriggerMsgPriority*
- *TriggerData*

The MQSET call has the same parameters as the MQINQ call. However for MQSET, all parameters except the completion code and reason code are input parameters. There are no partial-completion situations.

Note: You cannot use the MQI to set the attributes of WebSphere MQ objects other than locally-defined queues.

There is a description of the MQSET call in the *WebSphere MQ Application Programming Reference*.

Committing and backing out units of work

This chapter describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

The following terms, described below, are used in this topic:

- Commit
- Back out
- Syncpoint coordination
- Syncpoint
- Unit of work
- Single-phase commit
- Two-phase commit

If you are familiar with these transaction processing terms, you can skip to "Syncpoint considerations in WebSphere MQ applications" on page 184.

Commit and back out

When a program puts a message on a queue within a unit of work, that message is made visible to other programs only when the program *commits* the unit of work. To commit a unit of work, *all* updates must be successful to preserve data integrity. If the program detects an error and decides that the put operation should not be made permanent, it can *back out* the unit of work. When a program performs a backout, WebSphere MQ restores the queue by removing the messages that were put on the queue by that unit of work. The way in which the program performs the commit and back out operations depends on the environment in which the program is running.

Similarly, when a program gets a message from a queue within a unit of work, that message remains on the queue until the program commits the unit of work, but the message is not available to be retrieved by other programs. The message is permanently deleted from the queue when the program commits the unit of work. If the program backs out the unit of work, WebSphere MQ restores the queue by making the messages available to be retrieved by other programs.

Syncpoint coordination, syncpoint, unit of work

Syncpoint coordination is the process by which units of work are either committed or backed out with data integrity.

The decision to commit or back out the changes is taken, in the simplest case, at the end of a transaction. However, it can be more useful for an application to synchronize data changes at other logical points within a

transaction. These logical points are called *syncpoints* (or *synchronization points*) and the period of processing a set of updates between two syncpoints is called a *unit of work*. Several MQGET calls and MQPUT calls can be part of a single unit of work. The maximum number of messages within a unit of work can be controlled by the DEFINE MAXSMSGS command on z/OS, or by the MAXUMSGS attribute of the ALTER QMGR command on other platforms. See the *WebSphere MQ Script (MQSC) Command Reference* for details of these commands.

Single-phase commit
A *single-phase commit* process is one in which a program can commit updates to a queue without coordinating its changes with other resource managers.

Two-phase commit
A *two-phase commit* process is one in which updates that a program has made to WebSphere MQ queues can be coordinated with updates to other resources (for example, databases under the control of DB2). Under such a process, updates to *all* resources are committed or backed out together.

To help handle units of work, WebSphere MQ provides the *BackoutCount* attribute. This is incremented each time that a message within a unit of work is backed out. If the message repeatedly causes the unit of work to abend, the value of the *BackoutCount* finally exceeds that of the *BackoutThreshold*. This value is set when the queue is defined. In this situation, the application can remove the message from the unit of work and put it onto another queue, as defined in *BackoutRequeueQName*. When the message is moved, the unit of work can commit.

This chapter introduces committing and backing out units of work, under these headings:

Syncpoint considerations in WebSphere MQ applications

Two-phase commit is supported under:

- WebSphere MQ for AIX
- WebSphere MQ for i5/OS
- WebSphere MQ for HP-UX
- WebSphere MQ for Linux
- WebSphere MQ for Solaris
- WebSphere MQ for Windows
- CICS for MVS/ESA 4.1
- CICS Transaction Server for OS/390
- TXSeries
- IMS/ESA®
- z/OS batch with RRS
- Other external coordinators using the X/Open XA interface

Single-phase commit is supported under:

- WebSphere MQ for i5/OS
- WebSphere MQ on UNIX systems
- WebSphere MQ for Windows
- z/OS batch

Note: For further details on external interfaces see "Interfaces to external syncpoint managers" on page 191, and the XA documentation *CAE Specification Distributed Transaction Processing: The XA Specification*, published by The Open Group. Transaction managers (such as CICS, IMS, Encina, and Tuxedo) can participate in two-phase commit, coordinated with other recoverable resources. This means that the queuing functions provided by WebSphere MQ can be brought within the scope of a unit of work, managed by the transaction manager.

Samples shipped with WebSphere MQ show WebSphere MQ coordinating XA-compliant databases. For further information about these samples, see "Sample programs (all platforms except z/OS)" on page 395.

In your WebSphere MQ application, you can specify on every put and get call whether you want the call to be under syncpoint control. To make a put operation operate under syncpoint control, use the MQPMO_SYNCPOINT value in the *Options* field of the MQPMO structure when you call MQPUT. For a get operation, use the MQGMO_SYNCPOINT value in the *Options* field of the MQGMO structure. If you do not explicitly choose an option, the default action depends on the platform. The syncpoint control default on z/OS is yes; for all other platforms, it is no.

When your application receives an MQRC_BACKED_OUT reason code in response to an MQPUT or MQGET under syncpoint, the application should normally back out the current transaction using MBACK and then, if appropriate, retry the entire transaction. If the application receives MQRC_BACKED_OUT in response to an MQCMIT or MQDISC call, it does not need to call MQBACK.

Every time an MQGET call is backed out, the *BackoutCount* field of the MQMD structure of the affected message is incremented. A high *BackoutCount* indicates a message that has been repeatedly backed out. This might indicate a problem with this message, which you should investigate. See BackoutCount for details of *BackoutCount*.

Except on z/OS batch with RRS, if a program issues the MQDISC call while there are uncommitted requests, an implicit syncpoint occurs. If the program ends abnormally, an implicit backout occurs. On z/OS, an implicit syncpoint occurs if the program ends normally without first calling MQDISC.

For WebSphere MQ for z/OS programs, you can use the MQGMO_MARK_SKIP_BACKOUT option to specify that a message should not be backed out if backout occurs (in order to avoid an *MQGET-error-backout* loop). For information about using this option, see "Skipping backout" on page 152.

Changes to queue attributes (either by the MQSET call or by commands) are not affected by the committing or backing out of units of work.

Syncpoints in WebSphere MQ for z/OS applications

This section explains how to use syncpoints in transaction manager (CICS and IMS) and batch applications.

Syncpoints in CICS Transaction Server for OS/390 and CICS for MVS/ESA applications

In a CICS application you establish a syncpoint by using the EXEC CICS SYNCPOINT command.

To back out all changes to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command. For more information, see the *CICS Application Programming Reference*.

If other recoverable resources are involved in the unit of work, the queue manager (in conjunction with the CICS syncpoint manager) participates in a two-phase commit protocol; otherwise, the queue manager performs a single-phase commit process.

If a CICS application issues the MQDISC call, no implicit syncpoint is taken. If the application closes down normally, any open queues are closed and an implicit commit occurs. If the application closes down abnormally, any open queues are closed and an implicit backout occurs.

Syncpoints in IMS applications

In an IMS application, establish a syncpoint by using IMS calls such as GU (get unique) to the IOPCB and CHKP (checkpoint).

To back out all changes since the previous checkpoint, you can use the IMS ROLB (rollback) call. For more information, see the following books:

- *IMS/ESA Version 4 Application Programming: DL/I Calls*
- *IMS/ESA Version 4 Application Programming: Design Guide*
- *IMS/ESA Version 5 Application Programming: Database Manager*
- *IMS/ESA Version 5 Application Programming: Design Guide*

The queue manager (in conjunction with the IMS syncpoint manager) participates in a two-phase commit protocol if other recoverable resources are also involved in the unit of work.

All open handles are closed by the IMS adapter at a syncpoint (except in a nonmessage batch-oriented BMP). This is because a different user could initiate the next unit of work and WebSphere MQ security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made. The handles are closed at the beginning of the MQI call following the IMS call which initiated the syncpoint.

If you have not installed IMS APAR PN83757, handles are also closed after a ROLB call unless you are running IMS Version 3 or are running a nonmessage BMP.

If an IMS application (either a BMP or an MPP) issues the MQDISC call, open queues are closed but no implicit syncpoint is taken. If the application closes down normally, any open queues are closed and an implicit commit occurs. If the application closes down abnormally, any open queues are closed and an implicit backout occurs.

Syncpoints in z/OS batch applications

For batch applications, you can use the WebSphere MQ syncpoint management calls: MQCMIT and MQBACK. For backward compatibility, CSQBCMT and CSQBBAK are available as synonyms.

Note: If you need to commit or back out updates to resources managed by different resource managers, such as WebSphere MQ and DB2, within a single unit of work you can use RRS. For further information see "Transaction management and recoverable resource manager services."

Committing changes using the MQCMIT call:

As input, you must supply the connection handle (*Hconn*) that is returned by the MQCONN or MQCONNX call.

The output from MQCMIT is a completion code and a reason code. The call completes with a warning if the syncpoint was completed but the queue manager backed out the put and get operations since the previous syncpoint.

Successful completion of the MQCMIT call indicates to the queue manager that the application has reached a syncpoint and that all put and get operations made since the previous syncpoint have been made permanent.

Not all failure responses mean that the MQCMIT did not complete. For example, the application can receive MQRC_CONNECTION_BROKEN.

There is a description of the MQCMIT call in the *WebSphere MQ Application Programming Reference*.

Backing out changes using the MQBACK call:

As input, you must supply a connection handle (*Hconn*). Use the handle that is returned by the MQCONN or MQCONNX call.

The output from MQBACK is a completion code and a reason code.

The output indicates to the queue manager that the application has reached a syncpoint and that all gets and puts that have been made since the last syncpoint have been backed out.

There is a description of the MQBACK call in the *WebSphere MQ Application Programming Reference*.

Transaction management and recoverable resource manager services:

Transaction management and recoverable resource manager services (RRS) is a z/OS facility to provide two-phase syncpoint support across participating resource managers.

An application can update recoverable resources managed by various z/OS resource managers such as WebSphere MQ and DB2, and then commit or back out these updates as a single unit of work. RRS provides the necessary unit-of-work status logging during normal execution, coordinates the syncpoint processing, and provides appropriate unit-of-work status information during subsystem restart.

WebSphere MQ for z/OS RRS participant support enables WebSphere MQ applications in the batch, TSO, and DB2 stored procedure environments to update both WebSphere MQ and non-WebSphere MQ resources (for example, DB2) within a single logical unit of work. For information about RRS participant support, see *MVS Programming: Resource Recovery*.

Your WebSphere MQ application can use either MQCMIT and MQBACK or the equivalent RRS calls, SRRCMIT and SRRBACK. See "RRS batch adapter" on page 271 for more information.

RRS availability:

If RRS is not active on your z/OS system, any WebSphere MQ call issued from a program linked with either RRS stub (CSQBRSTB or CSQBRRSI) returns MQRC_ENVIRONMENT_ERROR.

DB2 stored procedures:

If you use DB2 stored procedures with RRS, be aware of the following:

- DB2 stored procedures that use RRS must be managed by workload manager (WLM-managed).
- If a DB2-managed stored procedure contains WebSphere MQ calls, and it is linked with either RRS stub (CSQBRSTB or CSQBRRSI), the MQCONN or MQCONNX call returns MQRC_ENVIRONMENT_ERROR.
- If a WLM-managed stored procedure contains WebSphere MQ calls, and is linked with a non-RRS stub, the MQCONN or MQCONNX call returns MQRC_ENVIRONMENT_ERROR, unless it is the first WebSphere MQ call executed since the stored procedure address space started.
- If your DB2 stored procedure contains WebSphere MQ calls and is linked with a non-RRS stub, WebSphere MQ resources updated in that stored procedure are not committed until the stored procedure address space ends, or until a subsequent stored procedure does an MQCMIT (using a WebSphere MQ Batch/TSO stub).
- Multiple copies of the same stored procedure can execute concurrently in the same address space. Ensure that your program is coded in a reentrant manner if you want DB2 to use a single copy of your stored procedure. Otherwise you might receive MQRC_HCONN_ERROR on any WebSphere MQ call in your program.
- Do not code MQCMIT or MQBACK in a WLM-managed DB2 stored procedure.
- Design all programs to run in Language Environment® (LE).

Syncpoints in CICS for i5/OS applications

WebSphere MQ for i5/OS participates in CICS for i5/OS units of work. You can use the MQI within a CICS for i5/OS application to put and get messages inside the current unit of work.

You can use the EXEC CICS SYNCPOINT command to establish a syncpoint that includes the WebSphere MQ for i5/OS operations. To back out all changes up to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command.

If you use MQPUT, MQPUT1, or MQGET with the MQPMO_SYNCPOINT, or MQGMO_SYNCPOINT, option set in a CICS for i5/OS application, you cannot log off CICS for i5/OS until WebSphere MQ for i5/OS has removed its registration as

an API commitment resource. Commit or back out any pending put or get operations before you disconnect from the queue manager. This allows you to log off CICS for i5/OS.

Syncpoints in WebSphere MQ for Windows, WebSphere MQ for i5/OS, and WebSphere MQ on UNIX systems

Syncpoint support operates on two types of units of work: local and global.

A *local* unit of work is one in which the only resources updated are those of the WebSphere MQ queue manager. Here syncpoint coordination is provided by the queue manager itself using a single-phase commit procedure.

A *global* unit of work is one in which resources belonging to other resource managers, such as databases, are also updated. WebSphere MQ can coordinate such units of work itself. They can also be coordinated by an external commitment controller such as another transaction manager or the i5/OS commitment controller.

For full integrity, use a two-phase commit procedure. Two-phase commit can be provided by XA-compliant transaction managers and databases such as IBM's TXSeries and UDB and also by the i5/OS commitment controller. WebSphere MQ products (except WebSphere MQ for i5/OS and WebSphere MQ for z/OS) can coordinate global units of work using a two-phase commit process. WebSphere MQ for i5/OS can act as a resource manager for global units of work within a WebSphere Application Server environment, but cannot act as a transaction manager.

Local units of work

Units of work that involve only the queue manager are called *local* units of work. Syncpoint coordination is provided by the queue manager itself (internal coordination) using a single-phase commit process.

To start a local unit of work, the application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. The unit of work is committed using MQCMIT or rolled back using MQBACK. However, the unit of work also ends when the connection between the application and the queue manager is broken, intentionally or unintentionally.

If an application disconnects (MQDISC) from a queue manager while a global unit of work coordinated by WebSphere MQ is still active, an attempt is made to commit the unit of work. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally.

Global units of work

Use global units of work when you also need to include updates to resources belonging to other resource managers. Here the coordination can be internal or external to the queue manager:

Internal syncpoint coordination:

Queue manager coordination of global units of work is not supported by WebSphere MQ for i5/OS or WebSphere MQ for z/OS. It is not supported in a WebSphere MQ client environment.

Here, WebSphere MQ does the coordination. To start a global unit of work, the application issues the MQBEGIN call.

As input to the MQBEGIN call, you must supply the connection handle (*Hconn*) that is returned by the MQCONN or MQCONNX call. This handle represents the connection to the WebSphere MQ queue manager.

The application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. This means that you can use MQBEGIN to initiate a global unit of work that updates local resources, resources belonging to other resource managers, or both. Updates made to resources belonging to other resource managers are made using the API of that resource manager. However, you cannot use the MQI to update queues that belong to other queue managers. Issue MQCMIT or MQBACK before starting further units of work (local or global).

The global unit of work is committed using MQCMIT; this initiates a two-phase commit of all the resource managers involved in the unit of work. A two-phase commit process is used whereby resource managers (for example, XA-compliant database managers such as DB2, Oracle, and Sybase) are firstly all asked to prepare to commit. Only if all are prepared are they asked to commit. If any resource manager signals that it cannot commit, each is asked to back out instead. Alternatively, you can use MQBACK to roll back the updates of all the resource managers.

If an application disconnects (MQDISC) while a global unit of work is still active, the unit of work is committed. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally.

The output from MQBEGIN is a completion code and a reason code.

When you use MQBEGIN to start a global unit of work, all the external resource managers that have been configured with the queue manager are included. However, the call starts a unit of work but completes with a warning if:

- There are no participating resource managers (that is, no resource managers have been configured with the queue manager)

or

- One or more resource managers are not available.

In these cases, the unit of work must include updates to only those resource managers that were available when the unit of work was started.

If one of the resource managers cannot commit its updates, all the resource managers are instructed to roll back their updates, and MQCMIT completes with a warning. In unusual circumstances (typically, operator intervention), an MQCMIT call might fail if some resource managers commit their updates but others roll them back; the work is deemed to have completed with a *mixed* outcome. Such occurrences are diagnosed in the error log of the queue manager so that remedial action may be taken.

An MQCMIT of a global unit of work succeeds if all the resource managers involved commit their updates.

For a description of the MQBEGIN call, see the *WebSphere MQ Application Programming Reference*.

External syncpoint coordination:

This occurs when a syncpoint coordinator other than WebSphere MQ has been selected; for example, CICS, Encina, or Tuxedo.

In this situation, WebSphere MQ on UNIX systems and WebSphere MQ for Windows register their interest in the outcome of the unit of work with the syncpoint coordinator so that they can commit or roll back any uncommitted get or put operations as required. The external syncpoint coordinator determines whether one- or two-phase commitment protocols are provided.

When you use an external coordinator, MQCMIT, MQBACK, and MQBEGIN cannot be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

The way in which an externally-coordinated unit of work is started depends on the programming interface provided by the syncpoint coordinator. An explicit call might be required. If an explicit call is required, and you issue an MQPUT call specifying the MQPMO_SYNCPOINT option when a unit of work is not started, the completion code MQRC_SYNCPOINT_NOT_AVAILABLE is returned.

The scope of the unit of work is determined by the syncpoint coordinator. The state of the connection between the application and the queue manager affects the success or failure of MQI calls that an application issues, not the state of the unit of work. An application can, for example, disconnect and reconnect to a queue manager during an active unit of work and perform further MQGET and MQPUT operations inside the same unit of work. This is known as a pending disconnect.

You can use WebSphere MQ API calls in CICS programs, whether or not you choose to use the XA abilities of CICS. If you do not use XA, then the puts and gets of messages to and from queues will not be managed within CICS atomic units of work. One reason for choosing this method is that the overall consistency of the unit of work is not very important to you.

If the integrity of your units of work is important to you, then you must use XA. When you use XA, CICS uses a two-phase commit protocol to ensure all resources within the unit of work are updated together.

For more information about setting up setting up transactional support, see the *WebSphere MQ System Administration Guide*, and also TXSeries CICS documentation, for example, *TXSeries for Multiplatforms CICS Administration Guide for Open Systems*.

Interfaces to external syncpoint managers

WebSphere MQ on UNIX systems, WebSphere MQ for i5/OS, and WebSphere MQ for Windows support coordination of transactions by external syncpoint managers that use the X/Open XA interface.

Some XA transaction managers (TXSeries) require that each XA resource manager supplies its name. This is the string called `name` in the XA switch structure. The resource manager for WebSphere MQ on UNIX and Windows systems is named MQSeries_XA_RMI. The name on i5/OS is MQSeries XA RMI For further details on XA interfaces refer to the XA documentation *CAE Specification Distributed Transaction Processing: The XA Specification*, published by The Open Group.

In an XA configuration, WebSphere MQ on UNIX systems and WebSphere MQ for Windows fulfil the role of an XA Resource Manager. An XA syncpoint coordinator

can manage a set of XA Resource Managers, and synchronize the commit or backout of transactions in both Resource Managers. This is how it works for a statically-registered resource manager:

1. An application notifies the syncpoint coordinator that it wants to start a transaction.
2. The syncpoint coordinator issues a call to any resource managers that it knows of, to notify them of the current transaction.
3. The application issues calls to update the resources managed by the resource managers associated with the current transaction.
4. The application requests that the syncpoint coordinator either commit or roll back the transaction.
5. The syncpoint coordinator issues calls to each resource manager using two-phase commit protocols to complete the transaction as requested.

The XA specification requires each Resource Manager to provide a structure called an *XA Switch*. This structure declares the capabilities of the Resource Manager, and the functions that are to be called by the syncpoint coordinator.

There are two versions of this structure:

MQRMIXASwitch	Static XA resource management
MQRMIXASwitchDynamic	Dynamic XA resource management

For a list of the libraries containing this structure see the *WebSphere MQ System Administration Guide*.

The method that must be used to link them to an XA syncpoint coordinator is defined by the coordinator; consult the documentation provided by that coordinator to determine how to enable WebSphere MQ to cooperate with your XA syncpoint coordinator.

The *xa_info* structure that is passed on any *xa_open* call by the syncpoint coordinator can be the name of the queue manager that is to be administered. This takes the same form as the queue manager name passed to MQCONN or MQCONNX, and can be blank if the default queue manager is to be used. However, you can use the two extra parameters TPM and AXLIB

TPM allows you to specify to WebSphere MQ the transaction manager name, for example, CICS. AXLIB allows you to specify the actual library name in the transaction manager where the XA AX entry points are located.

If you use either of these parameters or a non default queue manager you must specify the queue manager name using the QMNAME parameter. For further information see *WebSphere MQ Clients*.

Restrictions

1. Global units of work are not allowed with a shared Hconn (as described in "Shared (thread independent) connections with MQCONNX" on page 96.
2. WebSphere MQ for i5/OS does not support dynamic registration of XA resource managers.

   The only transaction manager supported is WebSphere Application Server.
3. On Windows systems, all functions declared in the XA switch are declared as _cdecl functions.

4. An external syncpoint coordinator can administer only one queue manager at a time. This is because the coordinator has an effective connection to each queue manager, and is therefore subject to the rule that only one connection is allowed at a time.
5. All applications that are run using the syncpoint coordinator can connect only to the queue manager that is administered by the coordinator because they are already effectively connected to that queue manager. They must issue MQCONN or MQCONNX to obtain a connection handle and must issue MQDISC before they exit. Alternatively, they can use the exit UE014015 for TXSeries CICS.

Interfaces to the i5/OS external syncpoint manager

WebSphere MQ for i5/OS can use native i5/OS commitment control as an external syncpoint coordinator.

Thread-independent (shared) connections are not allowed with commitment control. See the *i5/OS Programming: Backup and Recovery Guide, SC21-8079* for more information about the commitment control capabilities of i5/OS.

To start the i5/OS commitment control facilities, use the STRCMTCTL system command. To end commitment control, use the ENDCMTCTL system command.

Note: The default value of *Commitment definition scope* is *ACTGRP. This must be defined as *JOB for WebSphere MQ for i5/OS. For example:

```
STRCMTCTL LCKLVL(*ALL) CMTSCOPE(*JOB)
```

WebSphere MQ for i5/OS can also perform local units of work containing only updates to WebSphere MQ resources. The choice between local units of work and participation in global units of work coordinated byi5/OS is made in each application when the application calls MQPUT, MQPUT1, or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, or MQBEGIN. If commitment control is not active when the first such call is issued, WebSphere MQ starts a local unit of work and all further units of work for this connection to WebSphere MQ will also use local units of work, regardless of whether commitment control is subsequently started. To commit or back out a local unit of work, use MQCMIT or MQBACK respectively in the same way as other WebSphere MQ products. Thei5/OS commit and rollback calls such as the CL command COMMIT have no effect on WebSphere MQ local units of work.

If you want to use WebSphere MQ for i5/OS with native i5/OS commitment control as an external syncpoint coordinator, ensure that any job with commitment control is active and that you are using WebSphere MQ in a single-threaded job. If you call MQPUT, MQPUT1, or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, in a multithreaded job in which commitment control has been started, the call fails with a reason code of MQRC_SYNCPOINT_NOT_AVAILABLE.

It is possible to use local units of work and the MQCMIT and MQBACK calls in a multithreaded job.

If you call MQPUT, MQPUT1, or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, after starting commitment control, WebSphere MQ for i5/OS adds itself as an API commitment resource to the commitment definition.

This is typically the first such call in a job. While there are any API commitment resources registered under a particular commitment definition, you cannot end commitment control for that definition.

WebSphere MQ for i5/OS removes its registration as an API commitment resource when you disconnect from the queue manager, provided that there are no pending MQI operations in the current unit of work.

If you disconnect from the queue manager while there are pending MQPUT, MQPUT1, or MQGET operations in the current unit of work, WebSphere MQ for i5/OS remains registered as an API commitment resource so that it is notified of the next commit or rollback. When the next syncpoint is reached, WebSphere MQ for i5/OS commits or rolls back the changes as required. An application can disconnect and reconnect to a queue manager during an active unit of work and perform further MQGET and MQPUT operations inside the same unit of work (this is a pending disconnect).

If you attempt to issue an ENDCMTCTL system command for that commitment definition, message `CPF8355` is issued, indicating that pending changes were active. This message also appears in the job log when the job ends. To avoid this, commit or roll back all pending WebSphere MQ for i5/OS operations, and disconnect from the queue manager. Thus, using COMMIT or ROLLBACK commands before ENDCMTCTL should enable end-commitment control to complete successfully.

When you use i5/OS commitment control as an external syncpoint coordinator, you cannot issue MQCMIT, MQBACK, and MQBEGIN calls. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

To commit or roll back (that is, to back out) your unit of work, use one of the programming languages that supports the commitment control. For example:

- CL commands: COMMIT and ROLLBACK
- ILE C Programming Functions: _Rcommit and _Rrollback
- ILE RPG: COMMIT and ROLBK
- COBOL/400®: COMMIT and ROLLBACK

When you use i5/OS commitment control as an external syncpoint coordinator with WebSphere MQ for i5/OS, i5/OS performs a two-phase commit protocol in which WebSphere MQ participates. Because each unit of work is committed in two phases, the queue manager might become unavailable for the second phase after having voted to commit in the first phase. This can happen, for example, if the queue manager's internal jobs are ended. In this situation, the job log performing the commit contains message CPF835F indicating that a commit or rollback operation failed. The messages preceding this indicate the cause of the problem, whether it occurred during a commit or rollback operation, and also the logical unit of work ID (LUWID) for the failed unit of work.

If the problem was caused by the failure of the WebSphere MQ API commitment resource during the commit or rollback of a prepared unit of work, you can use the WRKMQMTRN command to complete the operation and restore the integrity of the transaction. The command requires that you know the LUWID of the unit of work to commit and back out.

Starting WebSphere MQ applications using triggers

Some WebSphere MQ applications that serve queues run continuously, so they are always available to retrieve messages that arrive on the queues. However,you might not want this when the number of messages arriving on the queues is unpredictable. In this case, applications could be consuming system resources even when there are no messages to retrieve.

WebSphere MQ provides a facility that enables an application to be started automatically when there are messages available to retrieve. This facility is known as *triggering*.

For information about triggering channels see WebSphere MQ Intercommunication.

This chapter introduces triggering, under these headings:

What is triggering?

The queue manager defines certain conditions as constituting *trigger events*.

If triggering is enabled for a queue and a trigger event occurs, the queue manager sends a *trigger message* to a queue called an *initiation queue*. The presence of the trigger message on the initiation queue indicates that a trigger event has occurred.

Trigger messages generated by the queue manager are not persistent. This has the effect of reducing logging (thereby improving performance), and minimizing duplicates during restart, so improving restart time.

The program that processes the initiation queue is called a *trigger-monitor application*, and its function is to read the trigger message and take appropriate action, based on the information contained in the trigger message. Usually this action is to start some other application to process the queue that generated the trigger message. From the point of view of the queue manager, there is nothing special about the trigger-monitor application; it is simply another application that reads messages from a queue (the initiation queue).

If triggering is enabled for a queue, you can create a *process-definition object* associated with it. This object contains information about the application that processes the message that caused the trigger event. If the process definition object is created, the queue manager extracts this information and places it in the trigger message, for use by the trigger-monitor application. The name of the process definition associated with a queue is given by the *ProcessName* local-queue attribute. Each queue can specify a different process definition, or several queues can share the same process definition.

If you want to trigger the start of a channel, you do not need to define a process definition object. The transmission queue definition is used instead.

Triggering is supported by WebSphere MQ clients running in the following environments:

- UNIX systems
- Windows systems

An application running in a client environment is the same as one running in a full WebSphere MQ environment, except that you link it with the client libraries. However the trigger monitor and the application to be started must both be in the same environment.

Triggering involves:

Application queue
An *application queue* is a local queue that, when it has triggering set on and when the conditions are met, requires that trigger messages are written.

Process definition
An application queue can have a *process definition object* associated with it that holds details of the application that will get messages from the application queue. (See the *WebSphere MQ Application Programming Reference* for a list of attributes.)

Remember that if you want a trigger to start a channel, you do not need to define a process definition object.

Transmission queue
You need a transmission queue if you want a trigger to start a channel.

For a transmission queue on AIX, HP-UX, i5/OS, Solaris, z/OS, or Windows systems, the `TriggerData` attribute of the transmission queue can specify the name of the channel to be started. This can replace the process definition for triggering channels, but is used only when a process definition is not created.

Trigger event
A *trigger event* is an event that causes a trigger message to be generated by the queue manager. This is usually a message arriving on an application queue, but it can also occur at other times (see "Conditions for a trigger event" on page 201). WebSphere MQ has a range of options to allow you to control the conditions that cause a trigger event (see "Controlling trigger events" on page 205).

Trigger message
The queue manager creates a *trigger message* when it recognizes a trigger event (see "Conditions for a trigger event" on page 201). It copies into the trigger message information about the application to be started. This information comes from the application queue and the process definition object associated with the application queue. Trigger messages have a fixed format (see "Format of trigger messages" on page 214).

Initiation queue
An *initiation queue* is a local queue on which the queue manager puts trigger messages. A queue manager can own more than one initiation queue, and each one is associated with one or more application queues. A shared queue, a local queue accessible by queue managers in a queue-sharing group, can be an initiation queue on WebSphere MQ for z/OS.

Trigger monitor

A *trigger monitor* is a continuously-running program that serves one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. The trigger monitor uses the information in the trigger message. It issues a command to start the application that is to retrieve the messages arriving on the application queue, passing it information contained in the trigger message header, which includes the name of the application queue.

On all platforms, a special trigger monitor known as the channel initiator is responsible for starting channels. On z/OS, the channel initiator is usually started manually, or it can be done automatically when a queue manager starts by changing CSQINP2 in the queue manager startup JCL. On other platforms, it is automatically started when the queue manager starts or it can be started manually with the runmqchi command.

(For more information, see "Trigger monitors" on page 209.)

To understand how triggering works, consider Figure 18, which is an example of trigger type FIRST (MQTT_FIRST).
In Figure 18, the sequence of events is:

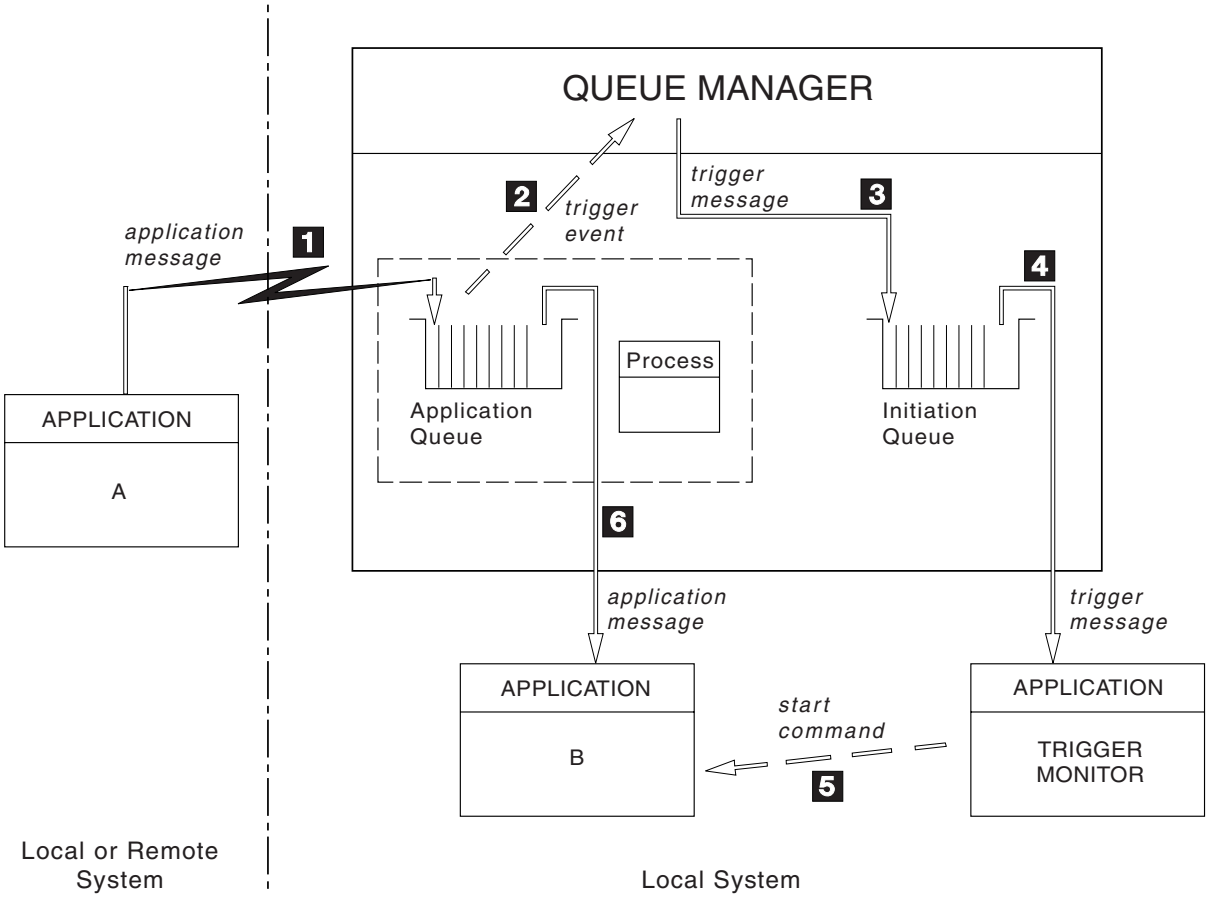


Figure 18. Flow of application and trigger messages

1. Application A, which can be either local or remote to the queue manager, puts a message on the application queue. No application has this queue open for input. However, this fact is relevant only to trigger type FIRST and DEPTH.

2. The queue manager checks to see if the conditions are met under which it has to generate a trigger event. They are, and a trigger event is generated. Information held within the associated process definition object is used when creating the trigger message.
3. The queue manager creates a trigger message and puts it on the initiation queue associated with this application queue, but only if an application (trigger monitor) has the initiation queue open for input.
4. The trigger monitor retrieves the trigger message from the initiation queue.
5. The trigger monitor issues a command to start application B (the server application).
6. Application B opens the application queue and retrieves the message.

Note:

1. If the application queue is open for input, by any program, and has triggering set for FIRST or DEPTH, no trigger event will occur because the queue is already being served.
2. If the initiation queue is not open for input, the queue manager does not generate any trigger messages; it waits until an application opens the initiation queue for input.
3. When using triggering for channels, use trigger type FIRST or DEPTH.

So far, the relationship between the queues within triggering has been only on a one to one basis. Consider Figure 19 on page 199.

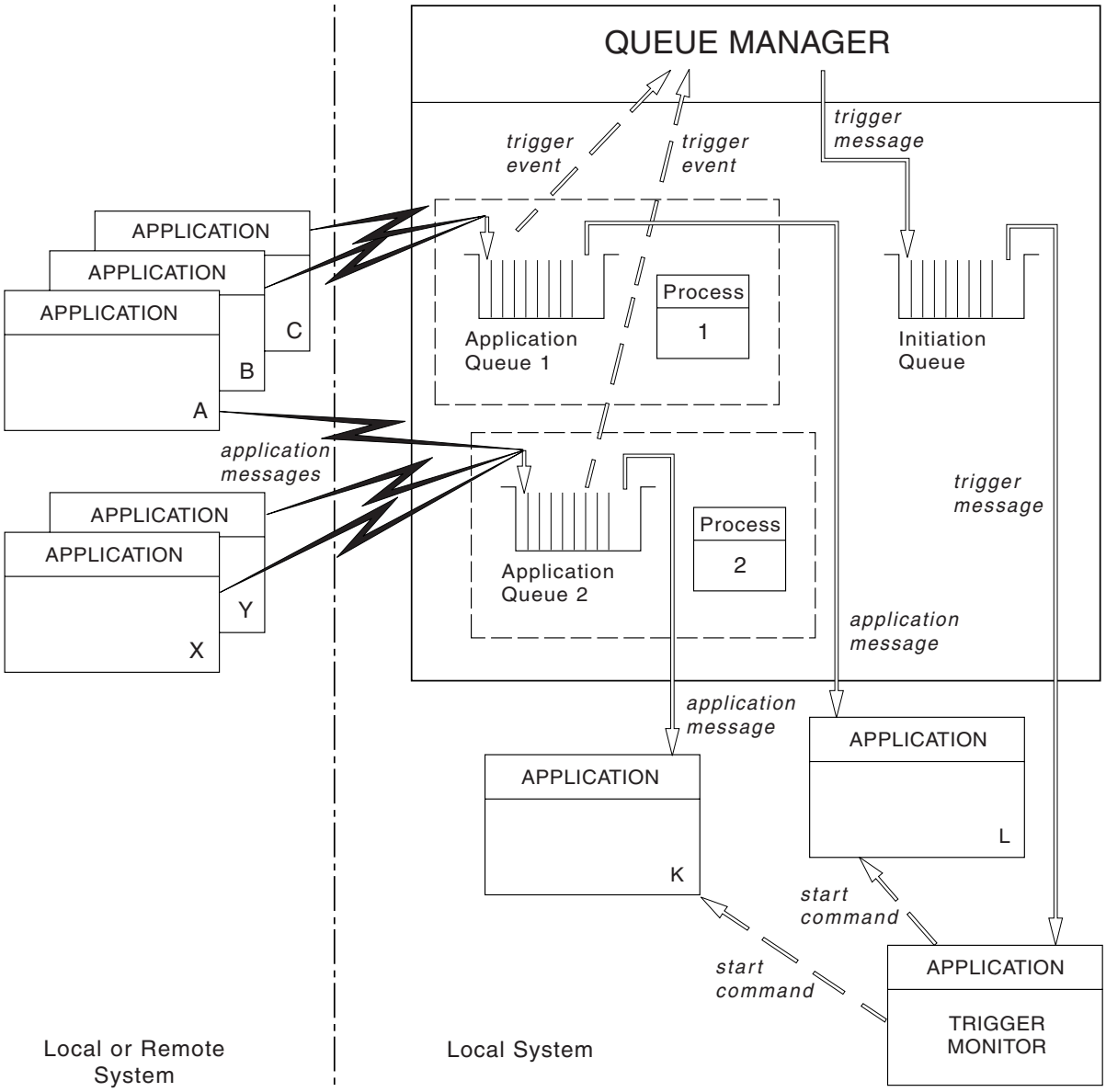


Figure 19. Relationship of queues within triggering

An application queue has a process definition object associated with it that holds details of the application that will process the message. The queue manager places the information in the trigger message, so only one initiation queue is necessary. The trigger monitor extracts this information from the trigger message and starts the relevant application to deal with the message on each application queue.

Remember that, if you want to trigger the start of a channel, you do not need to define a process definition object. The transmission queue definition can determine the channel to be triggered.

Prerequisites for triggering

Before your application can take advantage of triggering, follow the steps below:

1. Either:
   a. Create an initiation queue for your application queue. For example:

```
DEFINE QLOCAL (initiation.queue) REPLACE      +
       LIKE (SYSTEM.DEFAULT.LOCAL.QUEUE)      +
       DESCR ('initiation queue description')
```

or

b. Determine the name of a local queue that already exists and can be used by your application (usually this is SYSTEM.DEFAULT.INITIATION.QUEUE or, if you are starting channels with triggers, SYSTEM.CHANNEL.INITQ), and specify its name in the *InitiationQName* field of the application queue.

2. Associate the initiation queue with the application queue. A queue manager can own more than one initiation queue. You might want some of your application queues to be served by different programs, in which case, you can use one initiation queue for each serving program, although you do not have to. Here is an example of how to create an application queue:

```
DEFINE QLOCAL (application.queue) REPLACE   +
LIKE (SYSTEM.DEFAULT.LOCAL.QUEUE)           +
DESCR ('appl queue description')    +
INITQ ('initiation.queue')          +
PROCESS ('process.name')            +
TRIGGER                                     +
TRIGTYPE (FIRST)
```

Here is an extract from a CL program for WebSphere MQ for i5/OS that creates an initiation queue:

```
/*   Queue used by AMQSINQA                                        */
            CRTMQMQ     QNAME('SYSTEM.SAMPLE.INQ')                  +
                        QTYPE(*LCL)  REPLACE(*YES)                  +
                        MQMNAME                                     +
                        TEXT('queue for AMQSINQA')                  +
                        SHARE(*YES)                /* Shareable   */+
                        DFTMSGPST(*YES)/* Persistent messages OK  */+
                                                                    +
                        TRGENBL(*YES)  /* Trigger control on      */+
                        TRGTYPE(*FIRST)/* Trigger on first message*/+
                        PRCNAME('SYSTEM.SAMPLE.INQPROCESS')         +
                        INITQNAME('SYSTEM.SAMPLE.TRIGGER')
```

3. If you are triggering an application, create a process definition object to contain information relating to the application that is to serve your application queue. For example, to trigger-start a CICS payroll transaction called PAYR:

```
DEFINE PROCESS (process.name) +
       REPLACE +
       DESCR ('process description') +
       APPLTYPE ('CICS') +
       APPLICID ('PAYR') +
       USERDATA ('Payroll data')
```

Here is an extract from a CL program for WebSphere MQ for i5/OS that creates a process definition object:

```
/*   Process definition                                            */
            CRTMQMPRC   PRCNAME('SYSTEM.SAMPLE.INQPROCESS')         +
                        REPLACE(*YES)                               +
                        MQMNAME                                     +
                        TEXT('trigger process for AMQSINQA')        +
                        ENVDATA('JOBPTY(3)') /* Submit parameter  */+
                        APPID('AMQSINQA')    /* Program name      */
```

When the queue manager creates a trigger message, it copies information from the attributes of the process definition object into the trigger message.

Platform	To create a process definition object
UNIX systems, Windows systems	Use DEFINE PROCESS or use SYSTEM.DEFAULT.PROCESS and modify using ALTER PROCESS
z/OS	Use DEFINE PROCESS (see sample code in step 3 on page 200), or use the operations and control panels.
i5/OS	Use a CL program containing code as in step 3 on page 200.

4. Create a transmission queue definition and use blanks for the *ProcessName* attribute .

   The *TrigData* attribute can contain the name of the channel to be triggered or it can be left blank. Except on WebSphere MQ for z/OS, if it is left blank, the channel initiator searches the channel definition files until it finds a channel that is associated with the named transmission queue. When the queue manager creates a trigger message, it copies information from the *TrigData* attribute of the transmission queue definition into the trigger message.
5. If you have created a process definition object to specify properties of the application that is to serve your application queue, associate the process object with your application queue by naming it in the *ProcessName* attribute of the queue.

Platform	Use commands
UNIX systems, Windows systems	ALTER QLOCAL
z/OS	ALTER QLOCAL
i5/OS	CHGMQMQ

6. Start instances of the trigger monitors (or trigger servers in WebSphere MQ for i5/OS) that are to serve the initiation queues you have defined. See "Trigger monitors" on page 209 for more information.

If you want to be aware of any undelivered trigger messages, make sure that your queue manager has a dead-letter (undelivered-message) queue defined. Specify the name of the queue in the *DeadLetterQName* queue manager field.

You can then set the trigger conditions that you require, using the attributes of the queue object that defines your application queue. For more information on this, see "Controlling trigger events" on page 205.

Conditions for a trigger event

References to shared queues in this section mean shared queues in a queue-sharing group, only available on WebSphere MQ for z/OS.

The queue manager creates a trigger message when the following conditions are satisfied:

1. A message is *put* on a queue.
2. The message has a priority greater than or equal to the threshold trigger priority of the queue. This priority is set in the *TriggerMsgPriority* local queue attribute; if it is set to zero, any message qualifies.
3. The number of messages on the queue with priority greater than or equal to *TriggerMsgPriority* was previously, depending on *TriggerType*:
   - Zero (for trigger type MQTT_FIRST)

- Any number (for trigger type MQTT_EVERY)
- *TriggerDepth* minus 1 (for trigger type MQTT_DEPTH)

Note:

a. For non-shared local queues, the queue manager counts both committed and uncommitted messages when it assesses whether the conditions for a trigger event exist. Consequently an application might be started when there are no messages for it to retrieve because the messages on the queue have not been committed. In this situation, consider using the wait option with a suitable *WaitInterval*, so that the application waits for its messages to arrive.

b. For local shared queues the queue manager counts committed messages only.

4. For triggering of type FIRST or DEPTH, no program has the application queue open for removing messages (that is, the *OpenInputCount* local queue attribute is zero).

   Note:

   a. For shared queues, special conditions apply when multiple queue managers have trigger monitors running against a queue. In this situation, if one or more queue managers have the queue open for input shared, the trigger criteria on the other queue managers are treated as *TriggerType* MQTT_FIRST and *TriggerMsgPriority* zero. When all the queue managers close the queue for input, the trigger conditions revert to those specified in the queue definition.

   b. For shared queues, this condition is applied for each queue manager. That is, a queue manager's *OpenInputCount* for a queue must be zero for a trigger message to be generated for the queue by that queue manager. However, if any queue manager in the queue-sharing group has the queue open using the MQOO_INPUT_EXCLUSIVE option, no trigger message is generated for that queue by any of the queue managers in the queue-sharing group.

5. On WebSphere MQ for z/OS, if the application queue is one with a *Usage* attribute of MQUS_NORMAL, get requests for it are not inhibited (that is, the *InhibitGet* queue attribute is MQQA_GET_ALLOWED). Also, if the triggered application queue is one with a *Usage* attribute of MQUS_XMITQ, get requests for it are not inhibited.
6. Either:
   - The *ProcessName* local queue attribute for the queue is not blank, and the process definition object identified by that attribute has been created, or
   - The *ProcessName* local queue attribute for the queue is all blank, but the queue is a transmission queue. As the process definition is optional, the *TriggerData* attribute might also contain the name of the channel to be started. In this case, the trigger message contains attributes with the following values:
     - *QName*: queue name
     - *ProcessName*: blanks
     - *TriggerData*: trigger data
     - *ApplType*: MQAT_UNKNOWN
     - *ApplId*: blanks
     - *EnvData*: blanks
     - *UserData*: blanks

7. An initiation queue has been created, and has been specified in the *InitiationQName* local queue attribute. Also:
   - Get requests are not inhibited for the initiation queue (that is, the *InhibitGet* queue attribute is MQQA_GET_ALLOWED).
   - Put requests must not be inhibited for the initiation queue (that is, the *InhibitPut* queue attribute must be MQQA_PUT_ALLOWED).
   - The *Usage* attribute of the initiation queue must be MQUS_NORMAL.
   - In environments where dynamic queues are supported, the initiation queue must not be a dynamic queue that has been marked as logically deleted.
8. A trigger monitor currently has the initiation queue open for removing messages (that is, the *OpenInputCount* local queue attribute is greater than zero).
9. The trigger control (*TriggerControl* local queue attribute) for the application queue is set to MQTC_ON. To do this, set the *trigger* attribute when you define your queue, or use the ALTER QLOCAL command.
10. The trigger type (*TriggerType* local queue attribute) is not MQTT_NONE.

    If all the above required conditions are met, and the message that caused the trigger condition is put as part of a unit of work, the trigger message does not become available for retrieval by the trigger monitor application until the unit of work completes, whether the unit of work is committed *or*, for trigger type MQTT_FIRST or MQTT_DEPTH, backed out.
11. A suitable message is placed on the queue, for a *TriggerType* of MQTT_FIRST or MQTT_DEPTH, and the queue:
    - Was not previously empty (MQTT_FIRST), or
    - Had *TriggerDepth* or more messages (MQTT_DEPTH)

    and conditions 2 on page 201 through 10 (excluding 3 on page 201) are satisfied, if in the case of MQTT_FIRST a sufficient interval (*TriggerInterval* queue-manager attribute) has elapsed since the last trigger message was written for this queue.

    This is to allow for a queue server that ends before processing all the messages on the queue. The purpose of the trigger interval is to reduce the number of duplicate trigger messages that are generated.

    Note: If you stop and restart the queue manager, the *TriggerInterval timer* is reset. There is a small window during which it is possible to produce two trigger messages. The window exists when the queue's trigger attribute is set to enabled at the same time as a message arrives and the queue was not previously empty (MQTT_FIRST) or had *TriggerDepth* or more messages (MQTT_DEPTH).
12. The only application serving a queue issues an MQCLOSE call, for a *TriggerType* of MQTT_FIRST or MQTT_DEPTH, and there is at least:
    - One (MQTT_FIRST), or
    - *TriggerDepth* (MQTT_DEPTH)

    messages on the queue of sufficient priority (condition 2 on page 201), and conditions 6 on page 202 through 10 are also satisfied.

    This is to allow for a queue server that issues an MQGET call, finds the queue empty, and so ends; however, in the interval between the MQGET and the MQCLOSE calls, one or more messages arrive.

    Note:

a. If the program serving the application queue does not retrieve all the messages, this can cause a closed loop. Each time that the program closes the queue, the queue manager creates another trigger message that causes the trigger monitor to start the server program again.

b. If the program serving the application queue backs out its get request (or if the program abends) before it closes the queue, the same happens. However, if the program closes the queue before backing out the get request, and the queue is otherwise empty, no trigger message is created.

c. To prevent such a loop occurring, use the *BackoutCount* field of MQMD to detect messages that are repeatedly backed out. For more information, see "Messages that are backed out" on page 45.

13. The following conditions are satisfied using MQSET or a command:

a.

- *TriggerControl* is changed to MQTC_ON, or
- *TriggerControl* is already MQTC_ON and the value of either *TriggerType*, *TriggerMsgPriority*, or *TriggerDepth* (if relevant) is changed,

and there is at least:

- One (MQTT_FIRST or MQTT_EVERY), or
- *TriggerDepth* (MQTT_DEPTH)

messages on the queue of sufficient priority (condition 2 on page 201), and conditions 4 on page 202 through 10 on page 203 (excluding 8 on page 203) are also satisfied.

This is to allow for an application or operator changing the triggering criteria, when the conditions for a trigger to occur are already satisfied.

b. The *InhibitPut* queue attribute of an initiation queue changes from MQQA_PUT_INHIBITED to MQQA_PUT_ALLOWED, and there is at least:

- One (MQTT_FIRST or MQTT_EVERY), or
- *TriggerDepth* (MQTT_DEPTH)

messages of sufficient priority (condition 2 on page 201) on any of the queues for which this is the initiation queue, and conditions 4 on page 202 through 10 on page 203 are also satisfied. (One trigger message is generated for each such queue satisfying the conditions.)

This is to allow for trigger messages not being generated because of the MQQA_PUT_INHIBITED condition on the initiation queue, but this condition now having been changed.

c. The *InhibitGet* queue attribute of an application queue changes from MQQA_GET_INHIBITED to MQQA_GET_ALLOWED, and there is at least:

- One (MQTT_FIRST or MQTT_EVERY), or
- *TriggerDepth* (MQTT_DEPTH)

messages of sufficient priority (condition 2 on page 201) on the queue, and conditions 4 on page 202 through 10 on page 203, excluding 5 on page 202, are also satisfied.

This allows applications to be triggered only when they can retrieve messages from the application queue.

d. A trigger-monitor application issues an MQOPEN call for input from an initiation queue, and there is at least:

- One (MQTT_FIRST or MQTT_EVERY), or
- *TriggerDepth* (MQTT_DEPTH)

messages of sufficient priority (condition 2 on page 201) on any of the application queues for which this is the initiation queue, and conditions 4 on page 202 through 10 on page 203 (excluding 8 on page 203) are also satisfied, and no other application has the initiation queue open for input (one trigger message is generated for each such queue satisfying the conditions).

This is to allow for messages arriving on queues while the trigger monitor is not running, and for the queue manager restarting and trigger messages (which are nonpersistent) being lost.

14. MSGDLVSQ is set correctly. If you set MSGDLVSQ=FIFO, messages are delivered to the queue in a First In First Out basis. The priority of the message is ignored and the default priority of the queue is assigned to the message. If *TriggerMsgPriority* is set to a higher value than the default priority of the queue, no messages are triggered. If *TriggerMsgPriority* is set equal to or lower than the default priority of the queue, triggering occurs for type FIRST, EVERY, and DEPTH. For information about these types, see the description of the *TriggerType* field under "Controlling trigger events."

    If you set MSGDLVSQ=PRIORITY and the message priority is equal to or greater than the *TriggerMsgPriority* field, messages only *count* towards a trigger event. In this case, triggering occurs for type FIRST, EVERY, and DEPTH. As an example, if you put 100 messages of lower priority than the *TriggerMsgPriority*, the effective queue depth for triggering purposes is still zero. If you then put another message on the queue, but this time the priority is greater than or equal to the *TriggerMsgPriority*, the effective queue depth increases from zero to one and the condition for *TriggerType* FIRST is satisfied.

Note:

1. From step 12 on page 203 (where trigger messages are generated as a result of some event other than a message arriving on the application queue), the trigger message is not put as part of a unit of work. Also, if the *TriggerType* is MQTT_EVERY, and if there are one or more messages on the application queue, only one trigger message is generated.
2. If WebSphere MQ segments a message during MQPUT, a trigger event will not be processed until all the segments have been successfully placed on the queue. However, once message segments are on the queue, WebSphere MQ treats them as individual messages for triggering purposes. For example, a single logical message split into three pieces causes only one trigger event to be processed when it is first MQPUT and segmented. However, each of the three segments causes their own trigger events to be processed as they are moved through the WebSphere MQ network.

Controlling trigger events

You control trigger events using some of the attributes that define your application queue.

You can enable and disable triggering, and you can select the number or priority of the messages that count toward a trigger event. There is a full description of these attributes in the *WebSphere MQ Application Programming Reference*.

The relevant attributes are:

TriggerControl
: Use this attribute to enable and disable triggering for an application queue.

TriggerMsgPriority
: The minimum priority that a message must have for it to count toward a trigger event. If a message of priority less than *TriggerMsgPriority* arrives on the application queue, the queue manager ignores the message when it determines whether to create a trigger message. If *TriggerMsgPriority* is set to zero, all messages count toward a trigger event.

TriggerType
: In addition to the trigger type NONE (which disables triggering just like setting the *TriggerControl* to OFF), you can use the following trigger types to set the sensitivity of a queue to trigger events:

EVERY	A trigger event occurs every time that a message arrives on the application queue. Use this type of trigger if you want multiple instances of an application started. You must always process the queue until it is empty.
FIRST	A trigger event occurs only when the number of messages on the application queue changes from zero to one. Use this type of trigger if you want a serving program to start when the first message arrives on a queue, continue until there are no more messages to process, then end. Also see "Special case of trigger type FIRST" on page 207.
DEPTH	A trigger event occurs only when the number of messages on the application queue reaches the value of the *TriggerDepth* attribute. A typical use of this type of triggering is to start a program when all the replies to a set of requests are received. **Triggering by depth:** With triggering by depth, the queue manager disables triggering (using the <xph><pv>TriggerControl</pv></xph> attribute) after it creates a trigger message. Your application must re-enable triggering itself (by using the MQSET call) after this has happened. The action of disabling triggering is not under syncpoint control, so triggering cannot be re-enabled simply by backing out a unit of work. If a program backs out a put request that caused a trigger event, or if the program abends, you must re-enable triggering by using the MQSET call or the ALTER QLOCAL command.

TriggerDepth
: The number of messages on a queue that causes a trigger event when using triggering by depth.

The conditions that must be satisfied for a queue manager to create a trigger message are described in "Conditions for a trigger event" on page 201.

Example of the use of trigger type EVERY

Consider an application that generates requests for motor insurance. The application might send request messages to a number of insurance companies, specifying the same reply-to queue each time. It could set a trigger of type EVERY on this reply-to queue so that each time a reply arrives, the reply could trigger an instance of the server to process the reply.

Example of the use of trigger type FIRST

Consider an organization with a number of branch offices that each transmit details of the day's business to the head office. They all do this at the same time, at the end of the working day, and at the head office there is an application that processes the details from all the branch offices. The first message to arrive at the head office could cause a trigger event that starts this application. This application would continue processing until there are no more messages on its queue.

Example of the use of trigger type DEPTH

Consider a travel agency application that creates a single request to confirm a flight reservation, to confirm a reservation for a hotel room, to rent a car, and to order some travelers' checks. The application could separate these items into four request messages, sending each to a separate destination. It could set a trigger of type DEPTH on its reply-to queue (with the depth set to the value 4), so that it is restarted only when all four replies have arrived.

If another message (possibly from a different request) arrives on the reply-to queue before the last of the four replies, the requesting application is triggered early. To avoid this, when using DEPTH triggering to collect multiple replies to a request, always use a new reply-to queue for each request.

Special case of trigger type FIRST

With trigger type FIRST, if there is already a message on the application queue when another message arrives, the queue manager does not usually create another trigger message.

However, the application serving the queue might not actually open the queue (for example, the application might end, possibly because of a system problem). If an incorrect application name has been put into the process definition object, the application serving the queue will not pick up any of the messages. In these situations, if another message arrives on the application queue, there is no server running to process this message (and any other messages on the queue).

To deal with this, the queue manager creates further trigger messages under the following circumstances:

- If another message arrives on the application queue, but only if a predefined time interval has elapsed since the queue manager created the last trigger message for that queue. This time interval is defined in the queue manager attribute *TriggerInterval*. Its default value is 999 999 999 milliseconds.
- On WebSphere MQ for z/OS, application queues that name an open initiation queue are scanned periodically. If *TriggerInterval* milliseconds have passed since the last trigger message was sent and the queue satisfies the conditions for a trigger event and CURDEPTH is greater than zero, a trigger message is generated. This process is called backstop triggering.

Consider the following points when deciding on a value for the trigger interval to use in your application:

- If you set *TriggerInterval* to a low value, and there is no application serving the application queue, trigger type FIRST might behave like trigger type EVERY. This depends on the rate that messages are being put onto the application queue, which in turn might depend on other system activity. This is because, if the trigger interval is very small, another trigger message is generated each time

that a message is put onto the application queue, even though the trigger type is FIRST, not EVERY. (Trigger type FIRST with a trigger interval of zero is equivalent to trigger type EVERY.)

- On WebSphere MQ for z/OS if you set *TriggerInterval* to a low value, and there is no application serving the trigger type FIRST application queue, backstop triggering will generate a trigger message each time the periodic scan of application queues that name open initiation queues takes place.
- If a unit of work is backed out (see "Trigger messages and units of work") and the trigger interval has been set to a high value (or the default value), one trigger message is generated when the unit of work is backed out. However, if you have set the trigger interval to a low value or to zero (causing trigger type FIRST to behave like trigger type EVERY) many trigger messages can be generated. If the unit of work is backed out, all the trigger messages are still made available. The number of trigger messages generated depends on the trigger interval, the maximum number being reached when trigger interval has been set to zero.

Designing an application that uses triggered queues

You have seen how to set up, and control, triggering for your applications. Here are some tips to consider when you design your application.

Trigger messages and units of work

Trigger messages created because of trigger events that are not part of a unit of work are put on the initiation queue, outside any unit of work, with no dependence on any other messages, and are available for retrieval by the trigger monitor immediately.

Trigger messages created because of trigger events that *are* part of a unit of work are put on the initiation queue as part of the same unit of work. Trigger monitors cannot retrieve these trigger messages until the unit of work completes. This applies whether the unit of work is committed or backed out.

If the queue manager fails to put a trigger message on an initiation queue, it will be put on the dead-letter (undelivered-message) queue.

Note:

1. The queue manager counts both committed and uncommitted messages when it assesses whether the conditions for a trigger event exist.

   With triggering of type FIRST or DEPTH, trigger messages are made available even if the unit of work is backed out so that a trigger message is always available when the required conditions are met. For example, consider a put request within a unit of work for a queue that is triggered with trigger type FIRST. This causes the queue manager to create a trigger message. If another put request occurs, from another unit of work, this does not cause another trigger event because the number of messages on the application queue has now changed from one to two, which does not satisfy the conditions for a trigger event. Now if the first unit of work is backed out, but the second is committed, a trigger message is still created.

   However, this means that trigger messages are sometimes created when the conditions for a trigger event are *not* satisfied. Applications that use triggering must always be prepared to handle this situation. It is recommended that you use the wait option with the MQGET call, setting the *WaitInterval* to a suitable value.

2. For local shared queues (that is, shared queues in a queue-sharing group) the queue manager counts committed messages only.

Getting messages from a triggered queue

When you design applications that use triggering, be aware that there might be a delay between a trigger monitor starting a program and other messages becoming available on the application queue. This can happen when the message that causes the trigger event is committed before the others.

To allow time for messages to arrive, always use the wait option when you use the MQGET call to remove messages from a queue for which trigger conditions are set. The *WaitInterval* must be sufficient to allow for the longest reasonable time between a message being put and that put call being committed. If the message is arriving from a remote queue manager, this time is affected by:

- The number of messages that are put before being committed
- The speed and availability of the communication link
- The sizes of the messages

For an example of a situation where you should use the MQGET call with the wait option, consider the same example that we used when describing units of work. This was a put request within a unit of work for a queue that is triggered with trigger type FIRST. This event causes the queue manager to create a trigger message. If another put request occurs, from another unit of work, this does not cause another trigger event because the number of messages on the application queue has not changed from zero to one. Now if the first unit of work is backed out, but the second is committed, a trigger message is still created. So the trigger message is created at the time that the first unit of work is backed out. If there is a significant delay before the second message is committed, the triggered application might need to wait for it.

With triggering of type DEPTH, a delay can occur even if all relevant messages are eventually committed. Suppose that the *TriggerDepth* queue attribute has the value 2. When two messages arrive on the queue, the second causes a trigger message to be created. However, if the second message is the first to be committed, it is at that time that the trigger message becomes available. The trigger monitor starts the server program, but the program can retrieve only the second message until the first one is committed. So the program might need to wait for the first message to be made available.

Design your application so that it terminates if no messages are available for retrieval when your wait interval expires. If one or more messages arrive subsequently, rely on your application being re-triggered to process them. This method prevents applications being idle, and unnecessarily using resources.

Trigger monitors

To a queue manager, a trigger monitor is like any other application that serves a queue. However, a trigger monitor serves initiation queues.

A trigger monitor is usually a continuously-running program. When a trigger message arrives on an initiation queue, the trigger monitor retrieves that message. It uses information in the message to issue a command to start the application that is to process the messages on the application queue.

The trigger monitor must pass sufficient information to the program that it is starting so that the program can perform the right actions on the right application queue.

A channel initiator is an example of a special type of trigger monitor for message channel agents. In this situation however, you must use either trigger type FIRST or DEPTH.

Trigger monitors on z/OS

The following trigger monitor is provided for CICS Transaction Server for OS/390 and CICS for MVS/ESA:

CKTI You need to start one instance of CKTI for each initiation queue (see the *WebSphere MQ for z/OS System Administration Guide* for information on how to do this). CKTI passes the MQTM structure of the trigger message to the program that it starts by EXEC CICS START TRANSID. The started program gets this information by using the EXEC CICS RETRIEVE command. A program can use the EXEC CICS RETRIEVE command with the RTRANSID option to determine how the program was started; if the value returned is CKTI, the program was started by WebSphere MQ for z/OS. For an example of how to use CKTI, see the source code supplied for module CSQ4CVB2 in the Credit Check sample application supplied with WebSphere MQ for z/OS. See "The Credit Check sample" on page 485 for a full description.

The following trigger monitor is provided for IMS/ESA:

CSQQTRMN
You need to start one instance of CSQQTRMN for each initiation queue (see the *WebSphere MQ for z/OS System Administration Guide* for information on how to do this). CSQQTRMN passes the MQTMC2 structure of the trigger message to the programs that it starts.

Trigger monitors on UNIX and Windows systems

The following trigger monitors are provided for the server environment:

amqstrg0
This is a sample trigger monitor that provides a subset of the function provided by **runmqtrm**. See "Sample programs (all platforms except z/OS)" on page 395 for more information on amqstrg0.

runmqtrm
The syntax of this command is **runmqtrm** [*-m QMgrName*] [*-q InitQ*], where QMgrName is the queue manager and InitQ is the initiation queue. The default queue is SYSTEM.DEFAULT.INITIATION.QUEUE on the default queue manager. It calls programs for the appropriate trigger messages. This trigger monitor supports the default application type.

The command string passed by the trigger monitor to the operating system is built as follows:

1. The *ApplId* from the relevant PROCESS definition (if created)
2. The MQTMC2 structure, enclosed in quotation marks
3. The *EnvData* from the relevant PROCESS definition (if created)

where *ApplId* is the name of the program to run as it would be entered on the command line.

The parameter passed is the MQTMC2 character structure. A command string is invoked that has this string, exactly as provided, in quotation marks, in order that the system command will accept it as one parameter.

The trigger monitor does not look to see if there is another message on the initiation queue until the completion of the application that it has just started. If the application has a lot of processing to do, the trigger monitor might not be able to keep up with the number of trigger messages arriving. You have two options:

- Have more trigger monitors running
- Run the started applications in the background

If you have more trigger monitors running, you can control the maximum number of applications that can run at any one time. If you run applications in the background, there is no restriction imposed by WebSphere MQ on the number of applications that can run.

To run the started application in the background on Windows systems, within the *ApplId* field, prefix the name of your application with a START command. For example:

```
START /B AMQSECHA
```

To run the started application in the background on UNIX systems, put an & at the end of the *EnvData* of the PROCESS definition.

Note: Where a Windows path has spaces as a part of the path name, these should be enclosed in double quotes (") to ensure that it is handled as a single argument. For example, " C:\Program Files\Application Directory\Application.exe".

The following is an example of an APPLICID string where the file name includes spaces as a part of the path:

```
START "" /B "C:\Program Files\Application Directory\Application.exe"
```

The syntax of the Windows START command in the example includes an empty double-quoted string. START specifies that the first argument in double quotes will be treated as the title of the new command. To ensure that Windows does not mistake the application path for a 'title' argument, you should add a title string in double quotes to the command before the application name.

The following trigger monitors are provided for the WebSphere MQ client:

runmqtmc
This is the same as runmqtrm except that it links with the WebSphere MQ client libraries.

For CICS::

The amqltmc0 trigger monitor is provided for CICS. It works in the same way as the standard trigger monitor, runmqtrm, but you run it in a different way and it triggers CICS transactions.

It is supplied as a CICS program; define it with a 4-character transaction name. Enter the 4-character name to start the trigger monitor. It uses the default queue manager (as named in the qm.ini file or, on WebSphere MQ for Windows, the registry), and the SYSTEM.CICS.INITIATION.QUEUE.

If you want to use a different queue manager or queue, build the trigger monitor the MQTMC2 structure: this requires you to write a program using the EXEC CICS START call, because the structure is too long to add as a parameter. Then, pass the MQTMC2 structure as data to the START request for the trigger monitor.

When you use the MQTMC2 structure, you need to supply only the *StrucId*, *Version*, *QName*, and *QMgrName* parameters to the trigger monitor as it does not reference any other fields.

Messages are read from the initiation queue and used to start CICS transactions, using EXEC CICS START, assuming the APPL_TYPE in the trigger message is MQAT_CICS. The reading of messages from the initiation queue is performed under CICS syncpoint control.

Messages are generated when the monitor has started and stopped as well as when an error occurs. These messages are sent to the CSMT transient data queue.

Here are the available versions of the trigger monitor:

Version	Use
amqltmc0	TXSeries for AIX, HP-UX, and Sun Solaris Version 5.1
amqltmc4	TXSeries for Windows, Version 5.1

If you need a trigger monitor for other environments, write a program that can process the trigger messages that the queue manager puts on the initiation queues. Such a program should:

1. Use the MQGET call to wait for a message to arrive on the initiation queue.
2. Examine the fields of the MQTM structure of the trigger message to find the name of the application to start, and the environment in which it runs.
3. Issue an environment-specific start command. For example, in z/OS batch, submit a job to the internal reader.
4. Convert the MQTM structure to the MQTMC2 structure if required.
5. Pass either the MQTMC2 or MQTM structure to the started application. This can contain user data.
6. Associate with your application queue the application that is to serve that queue. You do this by naming the process definition object (if created) in the *ProcessName* attribute of the queue.

   Use DEFINE QLOCAL or ALTER QLOCAL. On i5/OS you can also use CRTMQMQ or CHGMQMQ.

For more information on the trigger monitor interface, see the *WebSphere MQ Application Programming Reference*.

WebSphere MQ for i5/OS trigger monitors

In WebSphere MQ for i5/OS, instead of **runmqtrm** (see "Trigger monitors on UNIX and Windows systems" on page 210), use the command

STRMQMTRM INITQNAME(*InitQ*) **MQMNAME**(*QMgrName*)

Details are as for runmqtrm.

The following sample programs are also provided, which you can use as models to write your own trigger monitors:

AMQSTRG4
This is a trigger monitor that submits an i5/OS job for the process that is to be started, but this means that there is a processing overhead associated with each trigger message.

AMQSERV4
This is a trigger server. For each trigger message, this server runs the command for the process in its own job, and can call CICS transactions.

Both the trigger monitor and the trigger server pass an MQTMC2 structure to the programs that they start. For a description of this structure, see the *WebSphere MQ Application Programming Reference*. Both of these samples are delivered in both source and executable forms.

Because these trigger monitors can invoke only native i5/OS programs, they cannot trigger Java programs directly, because Java classes are located in the IFS. However, Java programs can be triggered indirectly by triggering a CL program that then invokes the Java program and passes across the TMC2 structure. The minimum size of the TMC2 structure is 732 bytes.

The source of a sample CLP is shown below:

```
PGM PARM(&TMC2)
  DCL &TMC2 *CHAR LEN(800)
    ADDENVVAR ENVVAR(TM) VALUE(&TMC2)
    QSH  CMD('java_pgmname $TM')
    RMVENVVAR ENVVAR(TM)
ENDPGM
```

Properties of trigger messages

The following sections describe some other properties of trigger messages.

Persistence and priority of trigger messages

Trigger messages are not persistent because there is no requirement for them to be so.

However, the conditions for generating triggering events *do* persist, so trigger messages are generated whenever these conditions are met. In the event that a trigger message is lost, the continued existence of the application message on the application queue guarantees that the queue manager generates a trigger message as soon as all the conditions are met.

If a unit of work is rolled back, any trigger messages it generated are always delivered.

Trigger messages take the default priority of the initiation queue.

Queue manager restart and trigger messages

Following the restart of a queue manager, when an initiation queue is next opened for input, a trigger message can be put to this initiation queue if an application queue associated with it has messages on it, and is defined for triggering.

Trigger messages and changes to object attributes

Trigger messages are created according to the values of the trigger attributes in force at the time of the trigger event.

If the trigger message is not made available to a trigger monitor until later (because the message that caused it to be generated was put within a unit of work), any changes to the trigger attributes in the meantime have no effect on the trigger message. In particular, disabling triggering does not prevent a trigger message being made available once it has been created. Also, the application queue might no longer exist at the time that the trigger message is made available.

Format of trigger messages

The format of a trigger message is defined by the MQTM structure.

This has the following fields, which the queue manager fills when it creates the trigger message, using information in the object definitions of the application queue and of the process associated with that queue:

StrucId
: The structure identifier.

Version
: The version of the structure.

QName
: The name of the application queue on which the trigger event occurred. When the queue manager creates a trigger message, it fills this field using the *QName* attribute of the application queue.

ProcessName
: The name of the process definition object that is associated with the application queue. When the queue manager creates a trigger message, it fills this field using the *ProcessName* attribute of the application queue.

TriggerData
: A free-format field for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *TriggerData* attribute of the application queue. On any WebSphere MQ product except WebSphere MQ for z/OS, this field can be used to specify the name of the channel to be triggered.

ApplType
: The type of the application that the trigger monitor is to start. When the queue manager creates a trigger message, it fills this field using the *ApplType* attribute of the process definition object identified in *ProcessName*.

ApplId
: A character string that identifies the application that the trigger monitor is to start. When the queue manager creates a trigger message, it fills this field using the *ApplId* attribute of the process definition object identified in *ProcessName*. When you use trigger monitor CKTI or CSQQTRMN supplied by WebSphere MQ for z/OS, the *ApplId* attribute of the process definition object is a CICS or IMS transaction identifier.

EnvData
: A character field containing environment-related data for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *EnvData* attribute of the process definition object identified in *ProcessName*. The WebSphere MQ for z/OS-supplied trigger monitors (CKTI or CSQQTRMN) do not use this field, but other trigger monitors might choose to use it.

UserData
: A character field containing user data for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *UserData* attribute of the process definition object identified in *ProcessName*. This field can be used to specify the name of the channel to be triggered.

There is a full description of the trigger message structure in WebSphere MQ Application Programming Reference.

When triggering does not work

A program is not triggered if the trigger monitor cannot start the program or the queue manager cannot deliver the trigger message.

For example, the ApplId in the process object must specify that the program is to be started in the *background*; if this is not done, the trigger monitor cannot start the program.

If a trigger message is created but cannot be put on the initiation queue (for example, because the queue is full or the length of the trigger message is greater than the maximum message length specified for the initiation queue), the trigger message is put instead on the dead-letter (undelivered message) queue.

If the put operation to the dead-letter queue cannot complete successfully, the trigger message is discarded and a warning message is sent to the console (z/OS) or to the system operator (i5/OS), or put on the error log.

Putting the trigger message on the dead-letter queue might generate a trigger message for that queue. This second trigger message is discarded if it adds a message to the dead-letter queue.

If the program is triggered successfully but abends before it gets the message from the queue, use a trace utility (for example, CICS AUXTRACE if the program is running under CICS) to find the cause of the failure.

How CKTI detects errors

If the CKTI trigger monitor in WebSphere MQ for z/OS detects an error in the structure of a trigger message, or if it cannot start a program, it puts the trigger message on the dead-letter (undelivered message) queue.

CKTI adds a dead-letter header structure (MQDLH) to the trigger message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter queue.

An instance of CKTI stops serving an initiation queue if it attempts to get a trigger message from the queue and finds that the attributes of the queue have changed since it last accessed that queue. The attributes could have been changed by another program, or by an operator using the commands or operations and control panels of WebSphere MQ. CKTI produces an error message, which includes a reason code, explaining the action it has taken.

How CSQQTRMN detects errors

If the CSQQTRMN trigger monitor in WebSphere MQ for z/OS detects an error in the structure of a trigger message, or if it cannot start a program, it puts the trigger message on the dead-letter queue and sends a diagnostic message to a user specified LTERM (the default is MASTER). CSQQTRMN adds a dead-letter header structure (MQDLH) to the trigger message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter queue. If any other errors are detected, CSQQTRMN sends a diagnostic message to the specified LTERM, and then terminates.

How RUNMQTRM detects errors

Describes the actions taken by the RUNMQTRM trigger monitor when it detects certain errors.

In WebSphere MQ on UNIX systems, if the RUNMQTRM trigger monitor detects any of the following errors:

- Trigger message structure not valid
- Application type unsupported
- Program cannot start
- Data-conversion error

it puts the trigger message on the dead-letter queue, having added a dead-letter header structure (MQDLH) to the message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter queue.

In WebSphere MQ on Windows systems, the RUNMQTRM trigger monitor fails if the APPLTYPE in the process definition is a non-default application for Windows systems. For example, if an APPLTYPE of NOTESAGENT, which represents a Lotus Notes® agent, is specified in the process definition, the RUNMQTRM fails. To successfully trigger a non-default application type, you must write your own trigger monitor application.

Using and writing API exits

Introducing API exits

Not supported on WebSphere MQ for z/OS.

API exits let you write code that changes the behavior of WebSphere MQ API calls, such as MQPUT and MQGET, and then insert that code immediately before or immediately after those calls. Once you have written an exit program and identified it to WebSphere MQ, the queue manager automatically invokes your exit code at the registered points.

This chapter tells you how to write API exits, and how to set up WebSphere MQ to enable them. This section explains how you might use them and introduces the tasks involved. This chapter also contains the following major sections:

- "Compiling API exits" on page 218
- "Reference information" on page 223

Why use API exits

There are many reasons why you might want to insert code that modifies the behavior of applications at the level of the queue manager.

Each of your applications has a specific job to do, and its code should do that task as efficiently as possible. At a higher level, you might want to apply standards or business processes to a particular queue manager for *all* the applications that use that queue manager. It is more efficient to do this above the level of individual applications, and thus without having to change the code of each application affected.

Here are a few suggestions of areas in which API exits might be useful:

- For *security*, you can provide authentication, checking that applications are authorized to access a queue or queue manager. You can also police use of the API by applications by authenticating the individual API calls, or even the parameters that they use.
- For *flexibility*, you can respond to rapid changes in your business environment without changing the applications that rely on the data in that environment. You could, for example, have API exits that respond to changes in interest rates, currency exchange rates, or the price of components in a manufacturing environment.
- For *monitoring* use of a queue or queue manager, you can trace the flow of applications and messages, log errors in the API calls, set up audit trails for accounting purposes, or collect usage statistics for planning purposes.

How you use API exits

This section gives a brief overview of the tasks involved in setting up API exits. Each subsection here is supported by detailed information in the chapters in the rest of this information.

How to configure WebSphere MQ for API exits:

You configure WebSphere MQ to enable API exits either by:

- Using WebSphere MQ Explorer to add the IBM WebSphere MQ properties or the queue manager property, or
- Changing the Windows registry, or
- By editing the WebSphere MQ configuration files, mqs.ini and qm.ini, and adding new stanzas that:
  - Name the API exit
  - Identify the module and entry point of the API exit code to run
  - Optionally pass data with the exit
  - Identify the sequence of each exit in relation to other exits

For detailed information on this configuration, see the *WebSphere MQ System Administration Guide*. For a description of how API exits run, see "What happens when an API exit runs?" on page 218.

How to write an API exit:

You write exits using the C programming language. To help you to do so, we provide the source of a sample exit, amqsaxe0.c, that generates trace entries to a file that you specify. Use this as your starting point when writing exits.

Exits are available for every API call. Within API exits, the calls take the general form:

```
MQ_call_EXIT (parameters, context, ApiCallParameters)
```

where *call* is the MQI call name without the MQ prefix; for example, PUT, GET, and so on. The *parameters* control the function of the exit, *context* describes the context in which the API exit was called, and *ApiCallParameters* represent the parameters to the MQI call.

For more information about using the sample exit that we supply, see "The API exit sample program" on page 451. For reference information on the API exit calls, external control blocks, and associated topics, see "Reference information" on page 223.

What happens when an API exit runs?

The API exit routines to run are specified in stanzas in .ini files or in the Windows registry. You can specify an exit routine in three ways:

1. ApiExitCommon, in the mqs.ini file, identifies routines, for the whole of WebSphere MQ, applied when queue managers start up. These can be overridden by routines defined for individual queue managers (see item 3 in this list).
2. ApiExitTemplate, in the mqs.ini file, identifies routines, for the whole of WebSphere MQ, copied to the ApiExitLocal set (see item 3 in this list) when a new queue manager is created.
3. ApiExitLocal, in the qm.ini file, identifies routines that apply to a particular queue manager.

When a new queue manager is created, the ApiExitTemplate definitions in mqs.ini are copied to the ApiExitLocal definitions in qm.ini for the new queue manager. When a queue manager is started, both the ApiExitCommon and ApiExitLocal definitions are used. The ApiExitLocal definitions replace the ApiExitCommon definitions if both identify a routine of the same name. The `Sequence` attribute, described in the *WebSphere MQ System Administration Guide* determines the order in which the routines defined in the stanzas run.

Compiling API exits

Once you have written an exit, you compile and link it as follows.

The following examples show the commands used for the sample program described in "The API exit sample program" on page 451. For platforms other than Windows systems, you can find the sample API exit code in *install-dir*/samp and the compiled and linked shared library in *install-dir*/samp/bin. For Windows systems, you can find the sample API exit code in *install-dir*\Tools\c\Samples. *install-dir* is the directory in which WebSphere MQ was installed.

Note to users:

1. Guidance on programming 64 bit applications is listed in Chapter 10, "Coding standards on 64 bit platforms," on page 579

For information on configuring API exits, see the *WebSphere MQ System Administration Guide*.

On Solaris

SPARC platform:

32 bit applications:

Compile the API exit source code by issuing the following:

```
cc -xarch=v8plus -KPIC -mt -G -o /var/mqm/exits/amqsaxe \
   amqsaxe0.c -I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib \
   -R/usr/lib/32 -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket -lnsl -ldl
```

64 bit applications:

Compile the API exit source code by issuing the following:

```
cc -xarch=v9 -KPIC -mt -G -o /var/mqm/exits64/amqsaxe \
   amqsaxe0.c -I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 \
   -R/usr/lib/64  -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket -lnsl -ldl
```

x86-64 platform:

32 bit applications:

Compile the API exit source code by issuing the following:

```
cc -xarch=386 -KPIC -mt -G -o /var/mqm/exits/amqsaxe \
   amqsaxe0.c -I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib \
   -R/usr/lib/32 -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket \
   -lnsl -ldl
```

64 bit applications:

Compile the API exit source code by issuing the following:

```
cc -xarch=amd64 -KPIC -mt -G -o /var/mqm/exits64/amqsaxe \
   amqsaxe0.c -I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 \
   -R/usr/lib/64  -lmqm  -lmqmcs  -lmqmzse -lmqmzf -lsocket \
   -lnsl -ldl
```

On AIX

32 bit applications:

Non-threaded:

A file called amqsaxe.exp is supplied and contains the following:

```
#!
EntryPoint
MQStart
```

Compile the API exit source code by issuing the following command:

```
cc -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits/amqsaxe \
    amqsaxe0.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqmzf
```

Threaded:

A file called amqsaxe.exp is supplied and contains the following:

```
#!
EntryPoint
MQStart
```

Compile the API exit source code by issuing the following command:

```
xlc_r -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits/amqsaxe_r \
    amqsaxe0.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqmzf_r
```

64 bit applications:

Non-threaded:

A file called amqsaxe.exp is supplied and contains the following:

```
#!
EntryPoint
MQStart
```

Compile the API exit source code by issuing the following command:

```
cc -q64 -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits64/amqsaxe \
    amqsaxe0.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqmzf
```

Threaded:

A file called amqsaxe.exp is supplied and contains the following:

```
#!
EntryPoint
MQStart
```

Compile the API exit source code by issuing the following command:

```
xlc_r -q64 -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits64/amqsaxe_r \
    amqsaxe0.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqmzf_r
```

On HP-UX

PA-RISC platform:

32 bit applications:

Non-threaded:

1. Compile the ApiExit source code

   ```
   c89 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
   ```

2. Link the ApiExit source code

   ```
   ld +b: -b amqsaxe.o +ee MQStart -o  \
       /var/mqm/exits/amqsaxe -L/opt/mqm/lib -L/usr/lib -lmqmzf
   rm amqsaxe.o
   ```

Threaded:

1. Compile the ApiExit source code

   ```
   c89 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
   ```

2. Link the ApiExit object

   ```
   ld +b: -b amqsaxe.o +ee MQStart -o  \
      /var/mqm/exits/amqsaxe_r -L/opt/mqm/lib -L/usr/lib -lmqmzf_r -lpthread
   rm amqsaxe.o
   ```

64 bit applications:

Non-threaded:

1. Compile the ApiExit source code

   ```
    c89 +DD64 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
   ```

2. Link the ApiExit source code

   ```
   ld -b +noenvvar amqsaxe.o +ee MQStart \
       -o  /var/mqm/exits64/amqsaxe -L/opt/mqm/lib64 \
       -L/usr/lib/pa20_64 -lmqmzf
   rm amqsaxe.o
   ```

Threaded:

1. Compile the ApiExit source code

   ```
   c89 +DD64 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
   ```

2. Link the ApiExit object

   ```
   ld -b +noenvvar amqsaxe.o +ee MQStart  \
     -o  /var/mqm/exits64/amqsaxe_r -L/opt/mqm/lib64   \
     -L/usr/lib/pa20_64 -lmqmzf_r -lpthread
   rm amqsaxe.o
   ```

Itanium platform:

32 bit applications:

Non-threaded:

1. Compile the ApiExit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
```

2. Link the ApiExit source code

```
ld +b: -b amqsaxe.o +ee MQStart -o  \
    /var/mqm/exits/amqsaxe -L/opt/mqm/lib -L/usr/lib -lmqmzf
rm amqsaxe.o
```

Threaded:

1. Compile the ApiExit source code

```
c89 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
```

2. Link the ApiExit object

```
ld +b: -b amqsaxe.o +ee MQStart -o  \
    /var/mqm/exits/amqsaxe_r -L/opt/mqm/lib -L/usr/lib -lmqmzf_r -lpthread
rm amqsaxe.o
```

64 bit applications:

Non-threaded:

1. Compile the ApiExit source code

```
 c89 +DD64 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
```

2. Link the ApiExit source code

```
ld -b +noenvvar amqsaxe.o +ee MQStart \
    -o  /var/mqm/exits64/amqsaxe -L/opt/mqm/lib64 \
    -L/usr/lib/hpux64 -lmqmzf
rm amqsaxe.o
```

Threaded:

1. Compile the ApiExit source code

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o amqsaxe.o amqsaxe0.c -I/opt/mqm/inc
```

2. Link the ApiExit object

```
ld -b +noenvvar amqsaxe.o +ee MQStart  \
    -o  /var/mqm/exits64/amqsaxe_r -L/opt/mqm/lib64   \
    -L/usr/lib/hpux64 -lmqmzf_r -lpthread
rm amqsaxe.o
```

On Linux

31 bit applications (zSeries platform):

Non-threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/amqsaxe amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib \
    -Wl,-rpath=/usr/lib -lmqmzf
```

Threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/amqsaxe_r amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib \
    -Wl,-rpath=/usr/lib -lmqmzf_r
```

32 bit applications:

Non-threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/amqsaxe amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib \
    -Wl,-rpath=/usr/lib -lmqmzf
```

Threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/amqsaxe_r amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib \
    -Wl,-rpath=/usr/lib -lmqmzf_r
```

64 bit applications:

Non-threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsaxe amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64 \
    -Wl,-rpath=/usr/lib64 -lmqmzf
```

Threaded:

Compile the API exit source code by issuing the following command:

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsaxe_r amqsaxe0.c \
    -I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64 \
    -Wl,-rpath=/usr/lib64 -lmqmzf_r
```

On Windows systems

1. Create a file named amqsaxe.link containing the following:

   ```
   amqsaxe0.obj
   amqsaxe.exp
   mqm.lib mqmzf.lib
   msvcrt.lib oldnames.lib kernel32.lib user32.lib
   ```

   amqsaxe.def is provided.
2. Compile the API exit source code:

   ```
   set myccflags=-c -W3 -Gs- -Z7 -Od -nologo -LD -D_X86_=1
   set mydefines=-DWIN32 -D_WIN32 -D_MT -D_DLL
   cl %myccflags% %mydefines% amqsaxe0.c
   ```
3. Build the export file:

   ```
   lib -out:amqsaxe.lib  -def:amqsaxe.def  -machine:i386
   ```
4. Link the output from the compilation:

   ```
   link -nod -nologo -debug:full -dll @amqsaxe.link -out:amqsaxe.dll
   ```

On i5/OS

An exit is created as follows (for a C language example):

1. Create a module using CRTCMOD. Compile it to use teraspace by including the parameter `TERASPACE(*YES *TSIFC)`.
2. Create a service program from the module using CRTSRVPGM. You must bind it to the service program QMQM/LIBMQMZF_R for multithreaded API exits.

Reference information

This section contains reference information, mainly of interest to the programmer writing API exits.

It covers:

1. "External control blocks"
2. "The exit chain area and exit chain area header (MQACH)" on page 231
3. "External constants" on page 232
4. "C language typedefs" on page 234
5. "The exit entry point registration call (MQXEP)" on page 234
6. "Invoking exit functions" on page 237

External control blocks

Here we describe the structure of the external control blocks, MQAXP and MQAXC.

WebSphere MQ API exit parameter structure (MQAXP):

The MQAXP structure is used as an input/output parameter to the API exit.

MQAXP has the following C declaration:

```
typedef struct tagMQAXP {
  MQCHAR4   StrucId;           /* Structure identifier */
  MQLONG    Version;           /* Structure version number */
  MQLONG    ExitId;            /* Exit Identifier */
  MQLONG    ExitReason;        /* Exit invocation reason */
  MQLONG    ExitResponse;      /* Response code from exit */
  MQLONG    ExitResponse2;     /* Secondary response code from exit */
  MQLONG    Feedback;          /* Feedback code from exit */
  MQLONG    APICallerType;     /* MQSeries API caller type */
  MQBYTE16  ExitUserArea;      /* User area for use by exit */
  MQCHAR32  ExitData;          /* Exit data area */
  MQCHAR48  ExitInfoName;      /* Exit information name */
  MQBYTE48  ExitPDArea;        /* Problem determination area */
  MQCHAR48  QMgrName;          /* Name of local queue manager */
  PMQACH    ExitChainAreaPtr;  /* Inter exit communication area */
  MQHCONFIG Hconfig;           /* Configuration handle */
  MQLONG    Function;          /* Function Identifier */
};
```

The parameter list described below is passed when functions in an API exit are invoked:

StrucId (MQCHAR4) - input
The exit parameter structure identifier, with a value of:
`MQAXP_STRUC_ID.`

The exit handler sets this field on entry to each exit function.

Version (MQLONG) - input
The structure version number, with a value of:

MQAXP_VERSION_1
Version number for the exit parameter structure.

MQAXP_CURRENT_VERSION
Current version number for the exit parameter structure.

The exit handler sets this field on entry to each exit function.

ExitId (MQLONG) - input
The exit identifier, set on entry to the exit routine, indicating the type of exit:

MQXT_API_EXIT
API exit.

ExitReason (MQLONG) - input
The reason for invoking the exit, set on entry to each exit function:

MQXR_CONNECTION
The exit is being invoked to initialize itself before an MQCONN or MQCONNX call, or to end itself after an MQDISC call.

MQXR_BEFORE
The exit is being invoked before executing an API call, or before converting data on an MQGET.

MQXR_AFTER
The exit is being invoked after executing an API call.

ExitResponse (MQLONG) - output
The response from the exit, initialized on entry to each exit function to:

MQXCC_OK
Continue normally.

This field must be set by the exit function, to communicate to the queue manager the result of executing the exit function. The value must be one of the following:

MQXCC_OK
The exit function completed successfully. Continue normally.

This value can be set by all MQXR_* exit functions. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

MQXCC_FAILED
The exit function failed because of an error.

This value can be set by all MQXR_* exit functions. The queue manager sets CompCode to MQCC_FAILED, and Reason to:

- MQRC_API_EXIT_INIT_ERROR if the function is MQ_INIT_EXIT
- MQRC_API_EXIT_TERM_ERROR if the function is MQ_TERM_EXIT
- MQRC_API_EXIT_ERROR for all other exit functions

The values set can be altered by an exit function later in the chain.

ExitResponse2 is ignored; the queue manager continues processing as though MQXR2_SUPPRESS_CHAIN had been returned.

MQXCC_SUPPRESS_FUNCTION
Suppress WebSphere MQ API function.

This value can be set only by an MQXR_BEFORE exit function. It bypasses the API call. If it is returned by the MQ_DATA_CONV_ON_GET_EXIT, data conversion is bypassed. The queue manager sets CompCode to MQCC_FAILED, and Reason to MQRC_SUPPRESSED_BY_EXIT, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

If this value is set by an MQXR_AFTER or MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

MQXCC_SKIP_FUNCTION
Skip WebSphere MQ API function.

This value can be set only by an MQXR_BEFORE exit function. It bypasses the API call. If it is returned by the MQ_DATA_CONV_ON_GET_EXIT, data conversion is bypassed. The exit function must set CompCode and Reason to the values to be returned to the application, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is used to decide whether to invoke exit functions later in the chain.

If this value is set by an MQXR_AFTER or MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

MQXCC_SUPPRESS_EXIT
Suppress all exit functions belonging to the set of exits.

This value can be set only by the MQXR_BEFORE and MQXR_AFTER exit functions. It bypasses *all* subsequent invocations of exit functions belonging to this set of exits for this logical connection. This bypassing continues until the logical disconnect request occurs, when MQ_TERM_EXIT function is invoked with an ExitReason of MQXR_CONNECTION.

The exit function must set CompCode and Reason to the values to be returned to the application, but the values set can be altered by an exit function later in the chain. Other parameters for the call remain as the exit left them. ExitResponse2 is ignored.

If this value is set by an MQXR_CONNECTION exit function, the queue manager continues processing as though MQXCC_FAILED had been returned.

For information on the interaction between ExitResponse and ExitResponse2, and its affect on exit processing, see "How queue managers process exit functions" on page 228.

ExitResponse2 (MQLONG) - output
This is a secondary exit response code that qualifies the primary exit response code for MQXR_BEFORE exit functions. It is initialized to:

```
MQXR2_DEFAULT_CONTINUATION
```

on entry to a WebSphere MQ API call exit function. It can then be set to one of the values:

MQXR2_DEFAULT_CONTINUATION
Whether to continue with the next exit in the chain, depending on the value of ExitResponse.

If ExitResponse is MQXCC_SUPPRESS_FUNCTION or MQXCC_SKIP_FUNCTION, bypass exit functions later in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain. Invoke exit functions in the MQXR_AFTER chain that match exit functions earlier in the MQXR_BEFORE chain.

Otherwise, invoke the next exit in the chain.

MQXR2_SUPPRESS_CHAIN
Suppress the chain.

Bypass exit functions later in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain for this API call invocation. Invoke exit functions in the MQXR_AFTER chain that match exit functions earlier in the MQXR_BEFORE chain.

MQXR2_CONTINUE_CHAIN
Continue with the next exit in the chain.

For information on the interaction between ExitResponse and ExitResponse2, and its affect on exit processing, see "How queue managers process exit functions" on page 228.

Feedback (MQLONG) - input/output
Communicate feedback codes between exit function invocations. This is initialized to:

```
MQFB_NONE (0)
```

before invoking the first function of the first exit in a chain.

Exits can set this field to any value, including any valid MQFB_* or MQRC_* value. Exits can also set this field to a user-defined feedback value in the range MQFB_APPL_FIRST to MQFB_APPL_LAST.

APICallerType (MQLONG) - input
The API caller type, indicating whether the WebSphere MQ API caller is external or internal to the queue manager: MQXACT_EXTERNAL or MQXACT_INTERNAL.

ExitUserArea (MQBYTE16) - input/output
A user area, available to all the exits associated with a particular ExitInfoObject. It is initialized to MQXUA_NONE (binary zeros for the length of the ExitUserArea) before invoking the first exit function (MQ_INIT_EXIT) for the hconn. From then on, any changes made to this field by an exit function are preserved across invocations of functions of the same exit.

This field is aligned to a multiple of 4 MQLONGs.

Exits can also anchor any storage that they allocate from this area.

For each hconn, each exit in a chain of exits has a different ExitUserArea. The ExitUserArea cannot be shared by exits in a chain, and the contents of the ExitUserArea for one exit are not available to another exit in a chain.

For C programs, the constant MQXUA_NONE_ARRAY is also defined with the same value as MQXUA_NONE, but as an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH.

ExitData (MQCHAR32) - input
Exit data, set on input to each exit function to the 32 characters of exit-specific data that is provided in the exit. If you define no value in the exit this field is all blanks.

The length of this field is given by MQ_EXIT_DATA_LENGTH.

ExitInfoName (MQCHAR48) - input
The exit information name, set on input to each exit function to the ApiExit_name specified in the exit definitions in the stanzas.

ExitPDArea (MQBYTE48) - input/output
A problem determination area, initialized to `MQXPDA_NONE` (binary zeros for the length of the field) for each invocation of an exit function.

For C programs, the constant MQXPDA_NONE_ARRAY is also defined with the same value as MQXPDA_NONE, but as an array of characters instead of a string.

The exit handler always writes this area to the WebSphere MQ trace at the end of an exit, even when the function is successful.

The length of this field is given by MQ_EXIT_PD_AREA_LENGTH.

QMgrName (MQCHAR48) - input
The name of the local queue manager that has invoked an exit as a result of processing a WebSphere MQ API call.

If the name of a queue manager supplied on an MQCONN or MQCONNX calls is blank, this field is still set to the name of the local or default queue manager.

The exit handler sets this field on entry to each exit function.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH.

ExitChainAreaPtr (PMQACH) - input/output
This is used to communicate data across invocations of different exits in a chain. It is set to a NULL pointer before invoking the first function (MQ_INIT_EXIT with ExitReason MQXR_CONNECTION) of the first exit in a chain of exits. The value returned by the exit on one invocation is passed on to the next invocation.

Refer to “The exit chain area and exit chain area header (MQACH)” on page 231 for more details about how to use the exit chain area.

Hconfig (MQHCONFIG) - input
The configuration handle, representing the set of functions being initialized. This value is generated by the queue manager on the MQ_INIT_EXIT function, and is subsequently passed to the API exit function. It is set on entry to each exit function.

Function (MQLONG) - input
The function identifier, valid values for which are the MQXF_* constants described in “External constants” on page 232.

The exit handler sets this field to the correct value, on entry to each exit function, depending on the WebSphere MQ API call that resulted in the exit being invoked.

How queue managers process exit functions:

The processing performed by the queue manager on return from an exit function depends on both ExitResponse and ExitResponse2.

Table 11 below summarizes the possible combinations and their effects for an MQXR_BEFORE exit function, showing:

- Who sets the CompCode and Reason parameters of the API call
- Whether the remaining exit functions in the MQXR_BEFORE chain and the matching exit functions in the MQXR_AFTER chain are invoked
- Whether the API call is invoked

For an MQXR_AFTER exit function:

- CompCode and Reason are set in the same way as MQXR_BEFORE
- ExitResponse2 is ignored (the remaining exit functions in the MQXR_AFTER chain are always invoked)
- MQXCC_SUPPRESS_FUNCTION and MQXCC_SKIP_FUNCTION are not valid

For an MQXR_CONNECTION exit function:

- CompCode and Reason are set in the same way as MQXR_BEFORE
- ExitResponse2 is ignored
- MQXCC_SUPPRESS_FUNCTION, MQXCC_SKIP_FUNCTION, MQXCC_SUPPRESS_EXIT are not valid

In all cases, where an exit or the queue manager sets CompCode and Reason, the values set can be changed by an exit invoked later, or by the API call (if the API call is invoked later).

Table 11. MQXR_BEFORE exit processing

Value of ExitResponse	CompCode and Reason set by	Value of ExitResponse2 (default continuation) Chain	Value of ExitResponse2 (default continuation) API
MQXCC_OK	exit	Y	Y
MQXCC_SUPPRESS_EXIT	exit	Y	Y
MQXCC_SUPPRESS_FUNCTION	queue manager	N	N
MQXCC_SKIP FUNCTION	exit	N	N
MQXCC_FAILED	queue manager	N	N

WebSphere MQ API exit context structure (MQAXC):

The MQAXC structure is used as an input parameter to an API exit.

MQAXC has the following C declaration:

```
typedef struct tagMQAXC {
   MQCHAR4   StrucId;               /* Structure identifier */
   MQLONG    Version;               /* Structure version number */
   MQLONG    Environment;           /* Environment */
```

```
  MQCHAR12  UserId;                 /* UserId associated with appl */
  MQBYTE40  SecurityId              /* Extension to UserId running appl */
  MQCHAR264 ConnectionName;         /* Connection name */
  MQLONG    LongMCAUserIdLength;    /* long MCA user identifier length */
  MQLONG    LongRemoteUserIdLength; /* long remote user identifier length */
  MQPTR     LongMCAUserIdPtr;       /* long MCA user identifier address */
  MQPTR     LongRemoteUserIdPtr;    /* long remote user identifier address */
  MQCHAR28  ApplName;               /* Application name */
  MQLONG    ApplType;               /* Application type */
  MQPID     ProcessId;              /* Process identifier */
  MQTID     ThreadId;               /* Thread identifier */
};
```

The parameters to MQAXC are:

StrucId (MQCHAR4) - input
The exit context structure identifier, with a value of MQAXC_STRUC_ID. For C programs, the constant MQAXC_STRUC_ID_ARRAY is also defined, with the same value as MQAXC_STRUC_ID, but as an array of characters instead of a string.

The exit handler sets this field on entry to each exit function.

Version (MQLONG) - input
The structure version number, with a value of:

MQAXC_VERSION_1
Version number for the exit context structure.

MQAXC_CURRENT_VERSION
Current version number for the exit context structure.

The exit handler sets this field on entry to each exit function.

Environment (MQLONG) - input
The environment from which a WebSphere MQ API call was issued that resulted in an exit function being driven. Valid values for this field are:

MQXE_OTHER
An unrecognizable environment

MQXE_MCA
Message channel agent

MQXE_MCA_SVRCONN
A message channel agent acting on behalf of a client

MQXE_COMMAND_SERVER
The command server

MQXE_MQSC
The runmqsc command interpreter

The exit handler sets this field on entry to each exit function.

UserId (MQCHAR12) - input
The user ID associated with the application. In particular, in the case of client connections, this field contains the user ID of the adopted user as opposed to the user ID under which the channel code is running.

The exit handler sets this field on entry to each exit function. The length of this field is given by MQ_USER_ID_LENGTH.

SecurityId (MQBYTE40) - input
An extension to the userid running the application. Its length is given by MQ_SECURITY_ID_LENGTH.

ConnectionName (MQCHAR264) - input
The connection name field, set to the address of the client. For example, for TCP/IP, it would be the client IP address.

The length of this field is given by MQ_CONN_NAME_LENGTH.

LongMCAUserIdLength (MQLONG) - input
The length of the long MCA user identifier.

When MCA connects to the queue manager this field is set to the length of the long MCA user identifier (or zero if there is no such identifier).

LongRemoteUserIdLength (MQLONG) - input
The length of the long remote user identifier.

When MCA connects to the queue manager this field is set to the length of the long remote user identifier. Otherwise this field will be set to zero

LongMCAUserIdPtr (MQPTR) - input
Address of long MCA user identifier.

When MCA connects to the queue manager this field is set to the address of the long MCA user identifier (or to a null pointer if there is no such identifier).

LongRemoteUserIdPtr (MQPTR) - input
The address of the long remote user identifier.

When MCA connects to the queue manager this field is set to the address of the long remote user identifier (or to a null pointer if there is no such identifier).

ApplName (MQCHAR28) - input
The name of the application or component that issued the WebSphere MQ API call.

The rules for generating the ApplName are the same as for generating the default name for an MQPUT.

The value of this field is found by querying the operating system for the program name. Its length is given by MQ_APPL_NAME_LENGTH.

ApplType (MQLONG) - input
The type of application or component that issued the WebSphere MQ API call.

The value is MQAT_DEFAULT for the platform on which the application is compiled, or it equates to one of the defined MQAT_* values.

The exit handler sets this field on entry to each exit function.

ProcessId (MQPID) - input
The operating system process identifier.

Where applicable, the exit handler sets this field on entry to each exit function.

ThreadId (MQTID) - input
The MQ thread identifier. This is the same identifier used in MQ trace and FFST™ dumps, but might be different from the operating system thread identifier.

Where applicable, the exit handler sets this field on entry to each exit function.

The exit chain area and exit chain area header (MQACH)

If required, an exit function can acquire storage for an exit chain area and set the ExitChainAreaPtr in MQAXP to point to this storage.

Exits (either the same or different exit functions) can acquire multiple exit chain areas and link them together. Exit chain areas must only be added or removed from this list while called from the exit handler. This ensures that there are no serialization issues caused by different threads adding or removing areas from the list at the same time.

An exit chain area must start with an MQACH header structure, the C declaration for which is:

```
typedef struct tagMQACH {
   MQCHAR4   StrucId;             /* Structure identifier */
   MQLONG    Version;             /* Structure version number */
   MQLONG    StrucLength;         /* Length of the MQACH structure */
   MQLONG    ChainAreaLength;     /* Exit chain area length */
   MQCHAR48  ExitInfoName         /* Exit information name */
   PMQACH    NextChainAreaPtr;    /* Pointer to next exit chain area */
 };
```

The fields in the exit chain area header are:

StrucId (MQCHAR4) - input
The exit chain area structure identifier, with an initial value, defined by MQACH_DEFAULT, of MQACH_STRUC_ID.

For C programs, the constant MQACH_STRUC_ID_ARRAY is also defined; this has the same value as MQACH_STRUC_ID, but as an array of characters instead of a string.

Version (MQLONG) - input
The structure version number, as follows:

MQACH_VERSION_1
The version number for the exit parameter structure.

MQACH_CURRENT_VERSION
The current version number for the exit context structure.

The initial value of this field, defined by MQACH_DEFAULT, is `MQACH_CURRENT_VERSION`.

Note: If you introduce a new version of this structure, the layout of the existing part does not change. Exit functions must check that the version number is equal to or greater than the lowest version containing the fields that the exit function needs to use.

StrucLength (MQLONG) - input
The length of the MQACH structure. Exits can use this field to determine the start of the exit data, setting it to the length of the structure created by the exit.

The initial value of this field, defined by MQACH_DEFAULT, is MQACH_CURRENT_LENGTH.

ChainAreaLength (MQLONG) - input
The exit chain area length, set to the overall length of the current exit chain area, including the MQACH header.

The initial value of this field, defined by MQACH_DEFAULT, is zero.

ExitInfoName (MQCHAR48) - input
The exit information name.

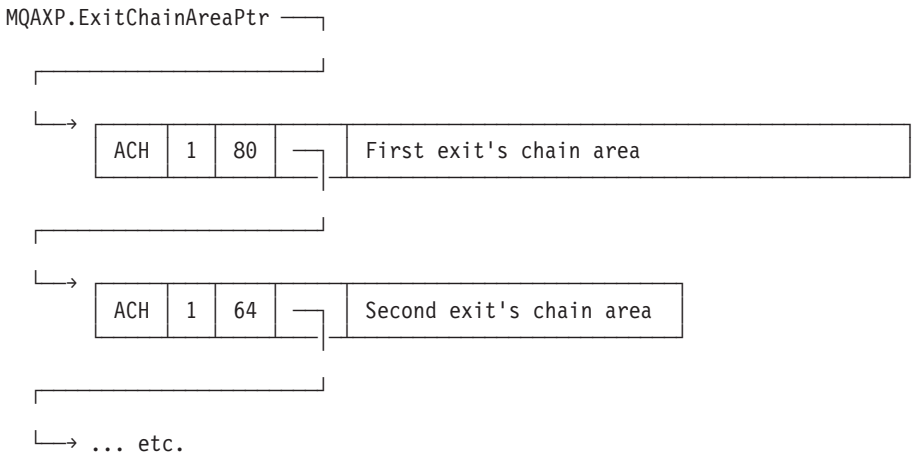
When an exit creates an MQACH structure, it must initialize this field with its own ExitInfoName, so that later this MQACH structure can be found by either another instance of this exit, or by a cooperating exit.

The initial value of this field, defined by MQACH_DEFAULT, is a zero length string ({""}).

NextChainAreaPtr (PMQACH) - input
A pointer to the next exit chain area with an initial value, defined by MQACH_DEFAULT, of null pointer (NULL).

Exit functions must release the storage for any exit chain areas that they acquire, and manipulate the chain pointers to remove their exit chain areas from the list.

An exit chain area can be constructed as follows:



External constants

The following external constants are available for API exits:

MQXF_* (exit function identifiers)

MQXF_INIT	1	X'00000001'
MQXF_TERM	2	X'00000002'
MQXF_CONN	3	X'00000003'
MQXF_CONNX	4	X'00000004'
MQXF_DISC	5	X'00000005'
MQXF_OPEN	6	X'00000006'
MQXF_CLOSE	7	X'00000007'
MQXF_PUT1	8	X'00000008'
MQXF_PUT	9	X'00000009'
MQXF_GET	10	X'0000000A'
MQXF_DATA_CONV_ON_GET	11	X'0000000B'
MQXF_INQ	12	X'0000000C'
MQXF_SET	13	X'0000000D'
MQXF_BEGIN	14	X'0000000E'
MQXF_CMIT	15	X'0000000F'
MQXF_BACK	16	X'00000010'

MQXR_* (exit reasons)

MQXR_BEFORE	1	X'00000001'
MQXR_AFTER	2	X'00000002'
MQXR_CONNECTION	3	X'00000003'

MQXE_* (environments)

```
MQXE_OTHER                      0       X'00000000'
MQXE_MCA                        1       X'00000001'
MQXE_MCA_SVRCONN                2       X'00000002'
MQXE_COMMAND_SERVER             3       X'00000003'
MQXE_MQSC                       4       X'00000004'
```

MQ*_* (additional constants)

```
MQAXP_VERSION_1                 1
MQAXP_VERSION_2                 2
MQAXC_VERSION_1                 1
MQACH_VERSION_1                 1

MQAXP_CURRENT_VERSION           1
MQAXC_CURRENT_VERSION           1
MQACH_CURRENT_VERSION           1

MQXACT_EXTERNAL                 1
MQXACT_INTERNAL                 2

MQXT_API_EXIT                   2

MQACH_LENGTH_1                  68 (32-bit platforms)
                                72 (64-bit platforms)
                                80 (128-bit platforms)

MQACH_CURRENT_LENGTH            68 (32-bit platforms)
                                72 (64-bit platforms)
                                80 (128-bit platforms)
```

MQ*_* (null constants)

```
MQXPDA_NONE                     X'00...00' (48 nulls)
MQXPDA_NONE_ARRAY               '\0','\0',...,'\0','\0'
```

MQXCC_* (completion codes)

```
MQXCC_FAILED                   -8
```

MQRC_* (reason codes)

MQRC_API_EXIT_ERROR 2374 X'00000946'
An exit function invocation has returned an invalid response code, or has failed in some way, and the queue manager cannot determine the next action to take.

Examine both the ExitResponse and ExitResponse2 fields of the MQAXP to determine the bad response code, and change the exit to return a valid response code.

MQRC_API_EXIT_INIT_ERROR 2375 X'00000947'
The queue manager encountered an error while initializing the execution environment for an API exit function.

MQRC_API_EXIT_TERM_ERROR 2376 X'00000948'
The queue manager encountered an error while closing the execution environment for an API exit function.

MQRC_EXIT_REASON_ERROR 2377 X'00000949'
The value of the ExitReason field supplied on an exit entry point registration call (MQXEP) call is in error.

Examine the value of the ExitReason field to determine and correct the bad exit reason value.

MQRC_RESERVED_VALUE_ERROR 2378 X'0000094A'
The value of the Reserved field is in error.

Examine the value of the Reserved field to determine and correct the Reserved value.

C language typedefs

Here are the C language typedefs associated with the API exits:

```
typedef PMQLONG  MQPOINTER PPMQLONG;
typedef PMQBYTE  MQPOINTER PPMQBYTE;
typedef PMQHOBJ  MQPOINTER PPMQHOBJ;
typedef PMQOD    MQPOINTER PPMQOD;
typedef PMQMD    MQPOINTER PPMQMD;
typedef PMQPMO   MQPOINTER PPMQPMO;
typedef PMQGMO   MQPOINTER PPMQGMO;
typedef PMQCNO   MQPOINTER PPMQCNO;
typedef PMQBO    MQPOINTER PPMQBO;

typedef MQAXP    MQPOINTER PMQAXP;
typedef MQACH    MQPOINTER PMQACH;
typedef MQAXC    MQPOINTER PMQAXC;

typedef MQCHAR   MQCHAR16[16];
typedef MQCHAR16 MQPOINTER PMQCHAR16;

typedef MQLONG   MQPID;
typedef MQLONG   MQTID;
```

The exit entry point registration call (MQXEP)

Use the MQXEP call to:

1. Register the *before* and *after* WebSphere MQ API exit invocation points at which to invoke exit functions
2. Specify the exit function entry points
3. Deregister the exit function entry points

You would usually code the MQXEP calls in the MQ_INIT_EXIT exit function, but you can specify them in any subsequent exit function.

If you use an MQXEP call to register an already registered exit function, the second MQXEP call completes successfully, replacing the registered exit function.

If you use an MQXEP call to register a NULL exit function, the MQXEP call will complete successfully and the exit function is deregistered.

If MQXEP calls are used to register, deregister, and reregister a given exit function during the life of a connection request, the previously registered exit function is reactivated. Any storage still allocated and associated with this exit function instance is available for use by the exit's functions. (This storage is usually released during the invocation of the termination exit function).

The interface to MQXEP is:

```
MQXEP (Hconfig, ExitReason, Function, EntryPoint, &ExitOpts, &CompCode, &Reason)
```

where:

Hconfig (MQHCONFIG) – input
The configuration handle, representing the API exit that includes the set of functions being initialized. This value is generated by the queue manager immediately before invoking the MQ_INIT_EXIT function, and is passed in the MQAXP to each API exit function.

ExitReason (MQLONG) – input
The reason for which the entry point is being registered, from the following:

- Connection level initialization or termination (MQXR_CONNECTION)
- Before a WebSphere MQ API call (MQXR_BEFORE)
- After a WebSphere MQ API call (MQXR_AFTER)

Function (MQLONG) – input
The function identifier, valid values for which are the MQXF_* constants (see "External constants" on page 232).

EntryPoint (PMQFUNC) - input
The address of the entry point for the exit function to be registered. The value NULL indicates either that the exit function has not been provided, or that a previous registration of the exit function is being deregistered.

ExitOpts(MQXEPO)
API exits can specify options that control how API exits are registered. If a null pointer is specified for this field, the default values of the MQXEPO structure are assumed.

CompCode (MQLONG) - output
The completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed.

Reason (MQLONG) - output
The reason code that qualifies the completion code.

If the completion code is MQCC_OK:

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

If the completion code is MQCC_FAILED:

MQRC_HCONFIG_ERROR
(2280, X'8E8') The supplied configuration handle is not valid. Use the configuration handle from the MQAXP.

MQRC_EXIT_REASON_ERROR
(2377, X'949') The supplied exit function invocation reason is either not valid or is not valid for the supplied exit function identifier.

Either use one of the valid exit function invocation reasons (MQXR_* value), or use a valid function identifier and exit reason combination. (See Table 12 on page 236.)

MQRC_FUNCTION_ERROR
(2281, X'8E9') The supplied function identifier is not valid for API exit reason. The following table shows valid combinations of function identifiers and ExitReasons.

Table 12. Valid combinations of function identifiers and ExitReasons

Function	ExitReason
MQXF_INIT MQXF_TERM	MQXR_CONNECTION
MQXF_CONN MQXF_CONNX MQXF_DISC MQXF_OPEN MQXF_CLOSE MQXF_PUT1 MQXF_PUT MQXF_GET MQXF_INQ MQXF_SET MQXF_BEGIN MQXF_CMIT MQXF_BACK MQXCF_STAT MQXCF_CB MQXCF_CTL MQXCF_CALLBACK MQXCF_SUB MQXCF_SUBRQ	MQXR_BEFORE MQXR_AFTER
MQXF_DATA_CONV_ON_GET	MQXR_BEFORE

MQRC_RESOURCE_PROBLEM
(2102, X'836') An attempt to register or deregister an exit function has failed because of a resource problem.

MQRC_UNEXPECTED_ERROR
(2195, X'893') An attempt to register or deregister an exit function has failed unexpectedly.

MQRC_PROPERTY_NAME_ERROR
(2442, X'098A') Invalid ExitProperties name.

MQRC_XEPO_ERROR
(2507, X'09CB') Exit options structure not valid.

MQXEP C language invocation:

```
MQXEP (Hconfig, ExitReason, Function, EntryPoint, &ExitOpts, &CompCode, &Reason);
```

Declaration for parameter list:

```
MQHCONFIG       Hconfig;        /* Configuration handle */
MQLONG          ExitReason;     /* Exit reason */
MQLONG          Function;       /* Function identifier */
PMQFUNC         EntryPoint;     /* Function entry point */
MQXEPO          ExitOpts;       /* Options that control the action of MQXEP */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code qualifying completion
                                   code */
```

MQXEP C function prototype:

```
void MQXEP (
MQHCONFIG    Hconfig,        /* Configuration handle */
MQLONG       ExitReason,     /* Exit reason */
MQLONG       Function,       /* Function identifier */
```

```
PMQFUNC      EntryPoint,     /* Function entry point */
PMQXEPO      pExitOpts;      /* Options that control the action of MQXEP */
PMQLONG      pCompCode,      /* Address of completion code */
PMQLONG      pReason);       /* Address of reason code qualifying completion
                                code */
```

Invoking exit functions

This section tells you how to invoke the exit functions available.

The descriptions of the individual functions start at "The API exit functions" on page 238. This section begins with some general information to help you when using these function calls.

General rules for API exit routines:

The following general rules apply when invoking API exit routines:

- In all cases, API exit functions are driven before validating API call parameters, and before any security checks (in the case of MQCONN, MQCONNX, or MQOPEN).
- The values of fields input to and output from an exit routine are:
  - On input to a *before* WebSphere MQ API exit function, the value of a field can be set by the application program, or by a previous exit function invocation.
  - On output from a *before* WebSphere MQ API exit function, the value of a field can be left unchanged, or set to some other value by the exit function.
  - On input to an *after* WebSphere MQ API exit function, the value of a field can be the value set by the queue manager after processing the WebSphere MQ API call, or can be set to a value by a previous exit function invocation in the chain of exit functions.
  - On output from an *after* WebSphere MQ API call exit function, the value of a field can be left unchanged, or set to some other value by the exit function.
- Exit functions must communicate with the queue manager by using the ExitResponse and ExitResponse2 fields.
- The CompCode and Reason code fields communicate back to the application. The queue manager and exit functions can set the CompCode and Reason code fields.
- The MQXEP call returns new reason codes to the exit functions that call MQXEP. However, exit functions can translate these new reason codes to any existing reasons codes that existing and new applications can understand.
- Each exit function prototype has similar parameters to the API function with an extra level of indirection except for the CompCode and Reason.

The execution environment:

In general, all errors from exit functions are communicated back to the exit handler using the ExitResponse and ExitResponse2 fields in MQAXP.

These errors in turn are converted into MQCC_* and MQRC_* values and communicated back to the application in the CompCode and Reason fields. However, any errors encountered in the exit handler logic are communicated back to the application as MQCC_* and MQRC_* values in the CompCode and Reason fields.

If an MQ_TERM_EXIT function returns an error:

- The MQDISC call has already taken place

- There is no other opportunity to drive the *after* MQ_TERM_EXIT exit function (and thus perform exit execution environment cleanup)
- Exit execution environment cleanup is *not* performed

In other words, the exit cannot be unloaded as it might still be in use. Also, other registered exits further down in the exit chain for which the *before* exit was successful, will be driven in the reverse order.

Setting up the exit execution environment:

While processing an explicit MQCONN or MQCONNX call, exit handling logic sets up the exit execution environment before invoking the exit initialization function (MQ_INIT_EXIT). Exit execution environment setup involves loading the exit, acquiring storage for, and initializing exit parameter structures. The exit configuration handle is also allocated at this point.

If errors occur during this phase, the MQCONN or MQCONNX call fails with CompCode MQCC_FAILED and one of the following reason codes:

MQRC_API_EXIT_LOAD_ERROR
An attempt to load an API exit module has failed.

MQRC_API_EXIT_NOT_FOUND
An API exit function could not be found in the API exit module.

MQRC_STORAGE_NOT_AVAILABLE
An attempt to initialize the execution environment for an API exit function failed because insufficient storage was available.

MQRC_API_EXIT_INIT_ERROR
An error was encountered while initializing the execution environment for an API exit function.

Cleaning up the exit execution environment:

While processing an explicit MQDISC call, or an implicit disconnect request as a result of an application ending, exit handling logic might need to clean up the exit execution environment after invoking the exit termination function (MQ_TERM_EXIT), if registered.

Cleaning up the exit execution environment involves releasing storage for exit parameter structures, possibly deleting any modules previously loaded into memory.

If errors occur during this phase, an explicit MQDISC call fails with CompCode MQCC_FAILED and the following reason code (errors are not highlighted on implicit disconnect requests):

MQRC_API_EXIT_TERM_ERROR
An error was encountered while closing the execution environment for an API exit function. The exit should *not* return any failure from the MQDISC before or after the MQ_TERM* API exit function calls.

The API exit functions:

This collection of topics describes each of the exit functions and its parameters.

- "Backout - MQ_BACK_EXIT" on page 239
- "Begin - MQ_BEGIN_EXIT" on page 241

Backout - MQ_BACK_EXIT:

MQ_BACK_EXIT provides a backout exit function to perform *before* and *after* backout processing. Use function identifier MQXF_BACK with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* backout call exit functions.

The interface to this function is:

```
MQ_BACK_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_BACK_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_BACK_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Callback - MQ_CALLBACK_EXIT:

MQ_CALLBACK_EXIT provides a subscription request exit function to perform *before* and *after* callback processing. Use function identifier MQXF_CALLBACK with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* callback call exit functions.

The interface to this function is:

```
MQ_CALLBACK_EXIT (&ExitParms, &ExitContext, &Hconn, &pMsgDesc, &pGetMsgOpts, &pBuffer, &pMQCBContext
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure

ExitContext (MQAXC) - input/output
Exit context structure

Hconn (MQHCONN) - input/output
Connection handle

pMsgDesc
Message descriptor

pGetMsgOpts
Options that control the action of MQGET

pBuffer
Area to contain the message data

pMQCBContext
Context data for the callback

C language invocation - MQ_CALLBACK_EXIT:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQMD    pMsgDesc;       /* Message descriptor */
PMQGMO   pGetMsgOpts;    /* Options that define the operation of the consumer */
PMQVOID  pBuffer;        /* Area to contain the message data */
PMQCBC   pContext;       /* Context data for the callback */
```

The queue manager then logically calls the exit as follows:

```
MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pMsgDesc, &pGetMsgOpts, &pBuffer,
 &pContext);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CALLBACK_EXIT (
PMQAXP    pExitParms;    /* Exit parameter structure */
PMQAXC    pExitContext;  /* Exit context structure */
PMQHCONN  pHconn;        /* Connection handle */
PPMQMD    ppMsgDesc;     /* Message descriptor */
PPMQGMO   ppGetMsgOpts;  /* Options that define the operation of the consumer */
PPMQVOID  ppBuffer;      /* Area to conatin the message data */
PPMQCBC   ppContext;)     /* Context data for the callback */
```

Begin - MQ_BEGIN_EXIT:

MQ_BEGIN_EXIT provides a begin exit function to perform *before* and *after* MQBEGIN call processing. Use function identifier MQXF_BEGIN with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQBEGIN call exit functions.

The interface to this function is:

```
MQ_BEGIN_EXIT (&ExitParms, &ExitContext, &Hconn, &pBeginOptions, &CompCode,
               &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

pBeginOptions (PMQBO)- input/output
Pointer to begin options.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQBO    pBeginOptions;  /* Ptr to begin options */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_BEGIN_EXIT (&ExitParms, &ExitContext, &Hconn, &pBeginOptions, &CompCode,
               &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_BEGIN_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PPMQBO    ppBeginOptions,   /* Address of ptr to begin options */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Close - MQ_CLOSE_EXIT:

MQ_CLOSE_EXIT provides a close exit function to perform *before* and *after* MQCLOSE call processing. Use function identifier MQXF_CLOSE with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQCLOSE call exit functions.

The interface to this function is:

```
MQ_CLOSE_EXIT (&ExitParms, &ExitContext, &Hconn, &pHobj,
               &Options, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

pHobj (PMQHOBJ) - input
Pointer to object handle.

Options (MQLONG)- input/output
Close options.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQHCONN         Hconn;          /* Connection handle */
PMQHOBJ         pHobj;          /* Ptr to object handle */
MQLONG          Options;        /* Close options */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CLOSE_EXIT (&ExitParms, &ExitContext,&Hconn, &pHobj, &Options,
               &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CLOSE_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQHCONN        pHconn,         /* Address of connection handle */
PPMQHOBJ        ppHobj,         /* Address of ptr to object handle */
PMQLONG         pOptions,       /* Address of close options */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Commit - MQ_CMIT_EXIT:

MQ_CMIT_EXIT provides a commit exit function to perform *before* and *after* commit processing. Use function identifier MQXF_CMIT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* commit call exit functions.

If a commit operation fails, and the transaction is backed out, the MQCMIT call fails with MQCC_WARNING and MQRC_BACKED_OUT. These return and reason codes are passed into any *after* MQCMIT exit functions to give the exit functions an indication that the unit of work has been backed out.

The interface to this function is:

```
MQ_CMIT_EXIT (&ExitParms, &ExitContext, &Hconn, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CMIT_EXIT (&ExitParms, &ExitContext,&Hconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CMIT_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Connect and connect extension - MQ_CONNX_EXIT:

MQ_CONNX_EXIT provides:

- Connection exit function to perform *before* and *after* MQCONN processing
- Connection extension exit function to perform *before* and *after* MQCONNX processing

The same interface, as described below, is invoked for both MQCONN and MQCONNX call exit functions.

When the message channel agent (MCA) responds to an inbound client connection, the MCA can connect and make a number of WebSphere MQ API calls before the client state is fully known. These API calls call the API exit functions with the MQAXC based on the MCA program itself (for example in the UserId and ConnectionName fields of the MQAXC).

When the MCA responds to subsequent inbound client API calls, the MQAXC structure is based on the inbound client, setting the UserId and ConnectionName fields appropriately.

The queue manager name set by the application on an MQCONN or MQCONNX call is passed to the underlying connect call. Any attempt by a *before* MQ_CONNX_EXIT to change the name of the queue manager has no effect.

Use function identifiers MQXF_CONN and MQXF_CONNX with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQCONN and MQCONNX call exit functions.

An MQ_CONNX_EXIT exit called for reason MQXR_BEFORE *must not* issue any WebSphere MQ API calls, as the correct environment has not been set up at this time.

The interface to MQCONN and MQCONNX is identical:

```
MQ_CONNX_EXIT (&ExitParms, &ExitContext, &pQMgrName, &pConnectOpts,
               &pHconn, &CompCode, &Reason);
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

pQMgrName (PMQCHAR) - input
Pointer to the queue manager name supplied on the MQCONNX call. The exit must not change this name on the MQCONN or MQCONNX call.

pConnectOpts (PMQCNO) - input/output
Pointer to the options that control the action of the MQCONNX call.

See "MQCNO - Connect options" in the *WebSphere MQ Application Programming Reference* for details.

For exit function MQXF_CONN, pConnectOpts points to the default connect options structure (MQCNO_DEFAULT).

pHconn (PMQHCONN) - input
Pointer to the connection handle.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Warning (partial completion)

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
PMQCHAR         pQMgrName;      /* Ptr to Queue manager name */
PMQCNO          pConnectOpts;   /* Ptr to Connection options */
PMQHCONN        pHconn;         /* Ptr to Connection handle */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CONNX_EXIT (&ExitParms, &ExitContext, &pQMgrName, &pConnectOps,
               &pHconn, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CONNX_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PPMQCHAR        ppQMgrName,     /* Address of ptr to queue manager name */
PPMQCNO         ppConnectOpts,  /* Address of ptr to connection options */
PPMQHCONN       ppHconn,        /* Address of ptr to connection handle */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Control callback - MQ_CTL_EXIT:

MQ_CTL_EXIT provides a subscription request exit function to perform *before* and *after* control callback processing. Use function identifier MQXF_CTL with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* control callback call exit functions.

The interface to this function is:

```
MQ_CTL_EXIT (&Hconn, &Operation, &ControlOpts, &CompCode, &Reason)
```

where the parameters are:

Hconn (MQHCONN) - input/output
Connection handle.

Operation (MQLONG) input/output
The operation being processed on the callback defined for the specified object handle

ControlOpts (MQCTLO) input/output
Options that control the action of MQCTL

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation - MQ_CTL_EXIT:

The queue manager logically defines the following variables:

```
MQHCONN  Hconn;          /* Connection handle */
MQLONG   Operation;      /* Operation being processed */
MQCTLO   ControlOpts;    /* Options that control the action of MQCTL  */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_CTL_EXIT (&Hconn, &Operation, &ControlOpts, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_CTL_EXIT (
PMQHCONN  pHconn;          /* Address of connection handle */
PMQLONG   pOperation;      /* Address of operation being processed */
PMQCTLO   pControlOpts;    /* Address of options that control the action of MQCTL  */
PMQLONG   pCompCode;       /* Address of completion code */
PMQLONG   pReason;)         /* Address of reason code qualifying completion code */
```

Disconnect - MQ_DISC_EXIT:

MQ_DISC_EXIT provides a disconnect exit function to perform *before* and *after* MQDISC exit processing. Use function identifier MQXF_DISC with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQDISC call exit functions.

The interface to this function is

```
MQ_DISC_EXIT (&ExitParms, &ExitContext, &pHconn,
              &CompCode, &Reason);
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

pHconn (PMQHCONN) - input
Pointer to the connection handle.

For the before MQDISC call, the value of this field is one of:

- The connection handle returned on the MQCONN or MQCONNX call
- Zero, for environments where an environment-specific adapter has connected to the queue manager
- A value set by a previous exit function invocation

For the after MQDISC call, the value of this field is zero or a value set by a previous exit function invocation.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
PMQHCONN        pHconn;         /* Ptr to Connection handle */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_DISC_EXIT (&ExitParms, &ExitContext, &pHconn,
              &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_DISC_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PPMQHCONN       ppHconn,        /* Address of ptr to connection handle */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Get - MQ_GET_EXIT:

MQ_GET_EXIT provides a get exit function to perform *before* and *after* MQGET call processing.

There are two function identifiers:

1. Use MQXF_GET with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQGET call exit functions.
2. Use MQXF_DATA_CONV_ON_GET with exit reason MQXR_BEFORE to register a *before* MQGET data conversion exit function.

The interface to this function is:

```
MQ_GET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pGetMsgOpts, &BufferLength, &pBuffer, &pDataLength,
             &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

Hobj (MQHOBJ) - input/output
Object handle.

pMsgDesc (PMQMD) - input/output
Pointer to message descriptor.

pGetMsgOpts (PMQPMO) - input/output
Pointer to get message options.

BufferLength (MQLONG) - input/output
Message buffer length.

pBuffer (PMQBYTE) - input/output
Pointer to message buffer.

pDataLength (PMQLONG) - input/output
Pointer to data length field.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQHCONN         Hconn;          /* Connection handle */
MQHOBJ          Hobj;           /* Object handle */
PMQMD           pMsgDesc;       /* Ptr to message descriptor */
PMQPMO          pGetMsgOpts;    /* Ptr to get message options */
MQLONG          BufferLength;   /* Message buffer length */
PMQBYTE         pBuffer;        /* Ptr to message buffer */
PMQLONG         pDataLength;    /* Ptr to data length field */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_GET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pGetMsgOpts, &BufferLength, &pBuffer, &pDataLength,
             &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_GET_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQHCONN        pHconn,         /* Address of connection handle */
PMQHOBJ         pHobj,          /* Address of object handle */
PPMQMD          ppMsgDesc,      /* Address of ptr to message descriptor */
PPMQGMO         ppGetMsgOpts,   /* Address of ptr to get message options */
PMQLONG         pBufferLength,  /* Address of message buffer length */
PPMQBYTE        ppBuffer,       /* Address of ptr to message buffer */
PPMQLONG        ppDataLength,   /* Address of ptr to data length field */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Initialization - MQ_INIT_EXIT:

MQ_INIT_EXIT provides connection level initialization, indicated by setting ExitReason in MQAXP to MQXR_CONNECTION.

During the initialization, note the following:

- The MQ_INIT_EXIT function calls MQXEP to register the WebSphere MQ API verbs and the ENTRY and EXIT points in which it is interested.
- Exits do not need to intercept all the WebSphere MQ API verbs. Exit functions are invoked only if an interest has been registered.
- Storage that is to be used by the exit can be acquired while initializing it.
- If a call to this function fails, the MQCONN or MQCONNX call that invoked it also fails with a CompCode and Reason that depend on the value of the ExitResponse field in MQAXP.
- An MQ_INIT_EXIT exit must not issue WebSphere MQ API calls, because the correct environment has not been set up at this time.
- If an MQ_INIT_EXIT fails with MQXCC_FAILED, the queue manager returns from the MQCONN or MQCONNX call that called it with MQCC_FAILED and MQRC_API_EXIT_ERROR.
- If the queue manager encounters an error while initializing the API exit function execution environment before invoking the first MQ_INIT_EXIT, the queue manager returns from the MQCONN or MQCONNX call that invoked MQ_INIT_EXIT with MQCC_FAILED and MQRC_API_EXIT_INIT_ERROR.

The interface to MQ_INIT_EXIT is:

```
MQ_INIT_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

CompCode (MQLONG) - input/output
Pointer to completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Pointer to reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

The CompCode and Reason returned to the application depend on the value of the ExitResponse field in MQAXP.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_INIT_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_INIT_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Inquire - MQ_INQ_EXIT:

MQ_INQ_EXIT provides an inquire exit function to perform *before* and *after* MQINQ call processing. Use function identifier MQXF_INQ with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQINQ call exit functions.

The interface to this function is:

```
MQ_INQ_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

Hobj (MQHOBJ) - input
Object handle.

SelectorCount (MQLONG) - input
Count of selectors

pSelectors (PMQLONG) - input/output
Pointer to array of selector values.

IntAttrCount (MQLONG) - input
Count of integer attributes.

pIntAttrs (PMQLONG) - input/output
Pointer to array of integer attribute values.

CharAttrLength (MQLONG) - input/output
Character attributes array length.

pCharAttrs (PMQCHAR) - input/output
Pointer to character attributes array.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;       /* Exit parameter structure */
MQAXC    ExitContext;     /* Exit context structure */
MQHCONN  Hconn;           /* Connection handle */
MQHOBJ   Hobj;            /* Object handle */
MQLONG   SelectorCount;   /* Count of selectors */
PMQLONG  pSelectors;      /* Ptr to array of attribute selectors */
MQLONG   IntAttrCount;    /* Count of integer attributes */
PMQLONG  pIntAttrs;       /* Ptr to array of integer attributes */
MQLONG   CharAttrLength;  /* Length of char attributes array */
PMQCHAR  pCharAttrs;      /* Ptr to character attributes */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_INQ_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_INQ_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PMQHOBJ   pHobj,            /* Address of object handle */
PMQLONG   pSelectorCount,   /* Address of selector count */
PPMQLONG  ppSelectors,      /* Address of ptr to array of selectors */
PMQLONG   pIntAttrCount;    /* Address of count of integer attributes */
PPMQLONG  ppIntAttrs,       /* Address of ptr to array of integer attributes */
PMQLONG   pCharAttrLength,  /* Address of character attribute length */
PPMQCHAR  ppCharAttrs,      /* Address of ptr to character attributes array */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Open - MQ_OPEN_EXIT:

MQ_OPEN_EXIT provides an open exit function to perform *before* and *after* MQOPEN call processing. Use function identifier MQXF_OPEN with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQOPEN call exit functions.

The interface to this function is

```
MQ_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &Options,
              &pHobj, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

pObjDesc (PMQOD) - input/output
Pointer to object descriptor.

Options (MQLONG) - input/output
Open options.

pHobj (PMQHOBJ) - input
Pointer to object handle.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQHCONN         Hconn;          /* Connection handle */
PMQOD           pObjDesc;       /* Ptr to object descriptor */
MQLONG          Options;        /* Open options */
PMQHOBJ         pHobj;          /* Ptr to object handle */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_OPEN_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &Options,
              &pHobj, &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_OPEN_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQHCONN        pHconn,         /* Address of connection handle */
PPMQOD          ppObjDesc,      /* Address of ptr to object descriptor */
PMQLONG         pOptions,       /* Address of open options */
PPMQHOBJ        ppHobj,         /* Address of ptr to object handle */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Put - MQ_PUT_EXIT:

MQ_PUT_EXIT provides a put exit function to perform *before* and *after* MQPUT call processing. Use function identifier MQXF_PUT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQPUT call exit functions.

The interface to this function is:

```
MQ_PUT_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

Hobj (MQHOBJ) - input/output
Object handle.

pMsgDesc (PMQMD) - input/output
Pointer to message descriptor.

pPutMsgOpts (PMQPMO) - input/output
Pointer to put message options.

BufferLength (MQLONG) - input/output
Message buffer length.

pBuffer (PMQBYTE) - input/output
Pointer to message buffer.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQHCONN         Hconn;          /* Connection handle */
MQHOBJ          Hobj;           /* Object handle */
PMQMD           pMsgDesc;       /* Ptr to message descriptor */
PMQPMO          pPutMsgOpts;    /* Ptr to put message options */
MQLONG          BufferLength;   /* Message buffer length */
PMQBYTE         pBuffer;        /* Ptr to message data */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_PUT_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &pMsgDesc,
             &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_PUT_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQHCONN        pHconn,         /* Address of connection handle */
PMQHOBJ         pHobj,          /* Address of object handle */
PPMQMD          ppMsgDesc,      /* Address of ptr to message descriptor */
PPMQPMO         ppPutMsgOpts,   /* Address of ptr to put message options */
PMQLONG         pBufferLength,  /* Address of message buffer length */
PPMQBYTE        ppBuffer,       /* Address of ptr to message buffer */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Put1 - MQ_PUT1_EXIT:

MQ_PUT1_EXIT provides a *put one message only* exit function to perform *before* and *after* MQPUT1 call processing. Use function identifier MQXF_PUT1 with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQPUT1 call exit functions.

The interface to this function is:

```
MQ_PUT1_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &pMsgDesc,
              &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

pObjDesc (PMQOD) - input/output
Pointer to object descriptor.

pMsgDesc (PMQMD) - input/output
Pointer to message descriptor.

pPutMsgOpts (PMQPMO) - input/output
Pointer to put message options.

BufferLength (MQLONG) - input/output
Message buffer length.

pBuffer (PMQBYTE) - input/output
Pointer to message buffer.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQHCONN         Hconn;          /* Connection handle */
PMQOD           pObjDesc;       /* Ptr to object descriptor */
PMQMD           pMsgDesc;       /* Ptr to message descriptor */
PMQPMO          pPutMsgOpts;    /* Ptr to put message options */
MQLONG          BufferLength;   /* Message buffer length */
PMQBYTE         pBuffer;        /* Ptr to message data */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_PUT1_EXIT (&ExitParms, &ExitContext, &Hconn, &pObjDesc, &pMsgDesc,
              &pPutMsgOpts, &BufferLength, &pBuffer, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_PUT1_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQHCONN        pHconn,         /* Address of connection handle */
PPMQOD          ppObjDesc,      /* Address of ptr to object descriptor */
PPMQMD          ppMsgDesc,      /* Address of ptr to message descriptor */
PPMQPMO         ppPutMsgOpts,   /* Address of ptr to put message options */
PMQLONG         pBufferLength,  /* Address of message buffer length */
PPMQBYTE        ppBuffer,       /* Address of ptr to message buffer */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Set - MQ_SET_EXIT:

MQ_SET_EXIT provides an inquire exit function to perform *before* and *after* MQSET call processing. Use function identifier MQXF_SET with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQSET call exit functions.

The interface to this function is:

```
MQ_SET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttr, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

Hobj (MQHOBJ) - input
Object handle.

SelectorCount (MQLONG) - input
Count of selectors

pSelectors (PMQLONG) - input/output
Pointer to array of selector values.

IntAttrCount (MQLONG) - input
Count of integer attributes.

pIntAttrs (PMQLONG) - input/output
Pointer to array of integer attribute values.

CharAttrLength (MQLONG) - input/output
Character attributes array length.

pCharAttrs (PMQCHAR) - input/output
Pointer to character attribute values.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;       /* Exit parameter structure */
MQAXC    ExitContext;     /* Exit context structure */
MQHCONN  Hconn;           /* Connection handle */
MQHOBJ   Hobj;            /* Object handle */
MQLONG   SelectorCount;   /* Count of selectors */
PMQLONG  pSelectors;      /* Ptr to array of attribute selectors */
MQLONG   IntAttrCount;    /* Count of integer attributes */
PMQLONG  pIntAttrs;       /* Ptr to array of integer attributes */
MQLONG   CharAttrLength;  /* Length of char attributes array */
```

```
PMQCHAR  pCharAttrs;      /* Ptr to character attributes */
MQLONG   CompCode;        /* Completion code */
MQLONG   Reason;          /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_SET_EXIT (&ExitParms, &ExitContext, &Hconn, &Hobj, &SelectorCount,
             &pSelectors, &IntAttrCount, &pIntAttrs, &CharAttrLength,
             &pCharAttrs, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_SET_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PMQHOBJ   pHobj,            /* Address of object handle */
PMQLONG   pSelectorCount,   /* Address of selector count */
PPMQLONG  ppSelectors,      /* Address of ptr to array of selectors */
PMQLONG   pIntAttrCount;    /* Address of count of integer attributes */
PPMQLONG  ppIntAttrs,       /* Address of ptr to array of integer attributes */
PMQLONG   pCharAttrLength,  /* Address of character attribute length */
PPMQCHAR  ppCharAttrs,      /* Address of ptr to character attributes array */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Status - MQ_STAT_EXIT:

MQ_STAT_EXIT provides a status exit function to perform *before* and *after* MQSTAT call processing. Use function identifier MQXF_STAT with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* MQSTAT call exit functions.

The interface to this function is:

```
MQ_STAT_EXIT (&ExitParms, &ExitContext, &Hconn, &Type, &pStatus
              &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input
Connection handle.

Type (MQLONG) - input
Type of status information to retrieve.

pStatus (PMQSTS) - output
Pointer to status buffer.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language declaration:

Your exit must match the following C function prototype:

```
void MQENTRY MQ_STAT_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PMQLONG   pType             /* Address of status type */
PPMQSTS   ppStatus          /* Address of status buffer */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

Termination - MQ_TERM_EXIT:

MQ_TERM_EXIT provides connection level termination, registered with a function identifier of MQXF_TERM and ExitReason MQXR_CONNECTION. If registered, MQ_TERM_EXIT is called once for every disconnect request.

As part of the termination, storage no longer required by the exit can be released, and any clean up required can be performed.

If an MQ_TERM_EXIT fails with MQXCC_FAILED, the queue manager returns from the MQDISC that called it with MQCC_FAILED and MQRC_API_EXIT_ERROR.

If the queue manager encounters an error while terminating the API exit function execution environment after invoking the last MQ_TERM_EXIT, the queue manager returns from the MQDISC call that invoked MQ_TERM_EXIT with MQCC_FAILED and MQRC_API_EXIT_TERM_ERROR

The interface to this function is:

```
MQ_TERM_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED, the exit function can set the reason code field to any valid MQRC_* value.

The CompCode and Reason returned to the application depend on the value of the ExitResponse field in MQAXP.

C language invocation:

The queue manager logically defines the following variables:

```
MQAXP           ExitParms;      /* Exit parameter structure */
MQAXC           ExitContext;    /* Exit context structure */
MQLONG          CompCode;       /* Completion code */
MQLONG          Reason;         /* Reason code */
```

The queue manager then logically calls the exit as follows:

```
MQ_TERM_EXIT (&ExitParms, &ExitContext, &CompCode, &Reason)
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_TERM_EXIT (
PMQAXP          pExitParms,     /* Address of exit parameter structure */
PMQAXC          pExitContext,   /* Address of exit context structure */
PMQLONG         pCompCode,      /* Address of completion code */
PMQLONG         pReason);       /* Address of reason code qualifying
                                   completion code */
```

Register subscription - MQ_SUB_EXIT:

MQ_SUB_EXIT provides a subscription request exit function to perform *before* and *after* subscription reregistration processing. Use function identifier MQXF_SUB with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* subscription registrationcall exit functions.

The interface to this function is:

```
MQ_SUB_EXIT (&ExitParms, &ExitContext, &Hconn, &pSubDesc, &pHobj, &pHsub, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input/output
Connection handle.

pSubDesc - input/output
Array of attribute selectors.

pHobj - input/output
Object handle

pHsub (MQHOBJ) input/output
Subscription handle

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation - MQ_SUB_EXIT:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQSD    pSubDesc;       /* Subscription descriptor */
PMQHOBJ  pHobj;          /* Object Handle */
PMQHOBJ  pHsub;          /* Subscription handle */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pSubDesc, &pHobj, &pHsub,
 &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
PMQAXP    pExitParms;      /* Exit parameter structure */
PMQAXC    pExitContext;    /* Exit context structure */
PMQHCONN  pHconn;          /* Connection handle */
PPMQSD    ppSubDesc;       /* Subscription descriptor */
PPMQHOBJ  ppHobj;          /* Object Handle */
PPMQHOBJ  ppHsub;          /* Subscription handle */
PMQLONG   pCompCode;       /* Completion code */
PMQLONG   pReason;         /* Reason code qualifying completion code */
```

Subscription request - MQ_SUBRQ_EXIT:

MQ_SUBRQ_EXIT provides a subscription request exit function to perform *before* and *after* subscription request processing. Use function identifier MQXF_SUBRQ with exit reasons MQXR_BEFORE and MQXR_AFTER to register *before* and *after* subscription request call exit functions.

The interface to this function is:

```
MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pHsub, &Action, &pSubRqOpts, &CompCode, &Reason)
```

where the parameters are:

ExitParms (MQAXP) - input/output
Exit parameter structure.

ExitContext (MQAXC) - input/output
Exit context structure.

Hconn (MQHCONN) - input/output
Connection handle.

pHsub (MQHOBJ) input/output
Subscription handle

Action (MQLONG) input/output
Action

pSubRqOpts (MQSRO) input/output

CompCode (MQLONG) - input/output
Completion code, valid values for which are:

MQCC_OK
Successful completion.

MQCC_WARNING
Partial completion.

MQCC_FAILED
Call failed

Reason (MQLONG) - input/output
Reason code qualifying the completion code.

If the completion code is MQCC_OK, the only valid value is:

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

If the completion code is MQCC_FAILED or MQCC_WARNING, the exit function can set the reason code field to any valid MQRC_* value.

C language invocation - MQ_SUBRQ_EXIT:

The queue manager logically defines the following variables:

```
MQAXP    ExitParms;      /* Exit parameter structure */
MQAXC    ExitContext;    /* Exit context structure */
MQHCONN  Hconn;          /* Connection handle */
PMQLONG  pHsub;          /* Subscription handle */
MQLONG   Action;         /* Action */
PMQSRO   pSubRqOpts;     /* Subscription Request Options */
MQLONG   CompCode;       /* Completion code */
MQLONG   Reason;         /* Reason code qualifying completion code */
```

The queue manager then logically calls the exit as follows:

```
MQ_SUBRQ_EXIT (&ExitParms, &ExitContext, &Hconn, &pHsub, &Action, &pSubRqOpts,
 &CompCode, &Reason);
```

Your exit must match the following C function prototype:

```
void MQENTRY MQ_SUBRQ_EXIT (
PMQAXP    pExitParms,       /* Address of exit parameter structure */
PMQAXC    pExitContext,     /* Address of exit context structure */
PMQHCONN  pHconn,           /* Address of connection handle */
PPMQHOBJ  ppHsub;           /* Address of Subscription handle */
PMQLONG   pAction;          /* Address of Action */
PPMQSRO   ppSubRqOpts;      /* Address of Subscription Request Options */
PMQLONG   pCompCode,        /* Address of completion code */
PMQLONG   pReason);         /* Address of reason code qualifying completion
                               code */
```

General information on invoking exit functions

This section provides some general guidance to help you to plan your exits, particularly related to handling errors and unexpected events.

What happens when exits fail:

If an exit function abnormally terminates after a destructive, out of syncpoint, MQGET call but before the message has been passed to the application, the exit handler can recover from the failure and pass control to the application.

In this case, the message might be lost. This is similar to what happens when an application fails immediately after receiving a message from a queue.

The MQGET call might complete with MQCC_FAILED and MQRC_API_EXIT_ERROR.

If a *before* API call exit function terminates abnormally, the exit handler can recover from the failure and pass control to the application without processing the API call. In this event, the exit function must recover any resources that it owns.

If chained exits are in use, the *after* API call exits for any *before* API call exits that had successfully been driven can themselves be driven. The API call might fail with MQCC_FAILED and MQRC_API_EXIT_ERROR.

Example error handling for exit functions:

The following diagram shows the points (e*N*) at which errors can occur. It is only an example to show how exits behave and should be read together with the following table. In this example, two exit functions are invoked both before and after each API call to show the behavior with chained exits.

```
Application ErrPt     Exit function            API call
----------- -----     -------------            --------

Start

MQCONN  -->
            e1
                          MQ_INIT_EXIT
            e2
                  before MQ_CONNX_EXIT  1
            e3
                  before MQ_CONNX_EXIT  2
            e4
                                          -->  MQCONN
            e5
                  after  MQ_CONNX_EXIT  2
            e6
                  after  MQ_CONNX_EXIT  1
            e7
        <--
```

```
MQOPEN  -->
                    before MQ_OPEN_EXIT  1
             e8
                    before MQ_OPEN_EXIT  2
             e9
                                             -->  MQOPEN
             e10
                    after  MQ_OPEN_EXIT  2
             e11
                    after  MQ_OPEN_EXIT  1
             e12
          <--
MQPUT   -->
                    before MQ_PUT_EXIT   1
             e13
                    before MQ_PUT_EXIT   2
             e14
                                             -->  MQPUT
             e15
                    after  MQ_PUT_EXIT   2
             e16
                    after  MQ_PUT_EXIT   1
             e17
          <--
MQCLOSE -->
                    before MQ_CLOSE_EXIT 1
             e18
                    before MQ_CLOSE_EXIT 2
             e19
                                             -->  MQCLOSE
             e20
                    after  MQ_CLOSE_EXIT 2
             e21
                    after  MQ_CLOSE_EXIT 1
             e22
          <--
MQDISC  -->
                    before MQ_DISC_EXIT  1
             e23
                    before MQ_DISC_EXIT  2
             e24
                                             -->  MQDISC
             e25
                    after  MQ_DISC_EXIT  2
             e26
                    after  MQ_DISC_EXIT  1
             e27

          <--

end
```

The following table lists the actions to be taken at each error point. Only a subset of the error points have been covered, as the rules shown here can apply to all others. It is the actions that specify the intended behavior in each case.

Table 13. API exit errors and appropriate actions to take

ErrPt	Description	Actions
e1	Error while setting up environment setup.	1. Undo environment setup as required 2. Drive no exit functions 3. Fail MQCONN with MQCC_FAILED, MQRC_API_EXIT_LOAD_ERROR

Table 13. API exit errors and appropriate actions to take (continued)

ErrPt	Description	Actions
e2	MQ_INIT_EXIT function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Clean up environment 2. Fail MQCONN with MQCC_FAILED, MQRC_API_EXIT_INIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Clean up environment
e3	*Before* MQ_CONNX_EXIT 1 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive MQ_TERM_EXIT function 2. Clean up environment 3. Fail MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Drive MQ_TERM_EXIT function if required 3. Clean up environment if required
e4	*Before* MQ_CONNX_EXIT 2 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive *after* MQ_CONNX_EXIT 1 function 2. Drive MQ_TERM_EXIT function 3. Clean up environment 4. Fail MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Drive *after* MQ_CONNX_EXIT 1 function if exit not suppressed 3. Drive MQ_TERM_EXIT function if required 4. Clean up environment if required
e5	MQCONN call fails.	1. Pass MQCONN CompCode and Reason 2. Drive *after* MQ_CONNX_EXIT 2 function if the *before* MQ_CONNX_EXIT 2 succeeded and the exit is not suppressed 3. Drive *after* MQ_CONNX_EXIT 1 function if the *before* MQ_CONNX_EXIT 1 succeeded and the exit is not suppressed 4. Drive MQ_TERM_EXIT function 5. Clean up environment
e6	*After* MQ_CONNX_EXIT 2 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive *after* MQ_CONNX_EXIT 1 function 2. Complete MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Drive *after* MQ_CONNX_EXIT 1 function if required

Table 13. API exit errors and appropriate actions to take (continued)

ErrPt	Description	Actions
e7	*After* MQ_CONNX_EXIT 1 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED, complete MQCONN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*[1]
e8	*Before* MQ_OPEN_EXIT 1 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED, complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*[1]
e9	*Before* MQ_OPEN_EXIT 2 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive *after* MQ_OPEN_EXIT 1 function 2. Complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_*, act as for the values of MQXCC_* and MQXR2_*[1]
e10	MQOPEN call fails	1. Pass MQOPEN CompCode and Reason 2. Drive *after* MQ_OPEN_EXIT 2 function if exit not suppressed 3. Drive *after* MQ_OPEN_EXIT 1 function if exit not suppressed and if chained exits not suppressed
e11	*After* MQ_OPEN_EXIT 2 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive *after* MQ_OPEN_EXIT 1 function 2. Complete MQOPEN call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Drive *after* MQ_OPEN_EXIT 1 function if exit not suppressed
e25	*After* MQ_DISC_EXIT 2 function completes with: • MQXCC_FAILED • MQXCC_*	• For MQXCC_FAILED: 1. Drive *after* MQ_DISC_EXIT 1 function 2. Drive MQ_TERM_EXIT function 3. Clean up exit execution environment 4. Complete MQDISC call with MQCC_FAILED, MQRC_API_EXIT_ERROR • For MQXCC_* 1. Act as for the values of MQXCC_* and MQXR2_*[1] 2. Drive MQ_TERM_EXIT function 3. Clean up exit execution environment

Note:

1. The values of MQXCC_* and MQXR2_* and their corresponding actions are defined in "How queue managers process exit functions" on page 228.

What if the ExitResponse fields are incorrectly set:

If the ExitResponse field is set to a value other than one of the supported values, the following actions apply:

- For a *before* MQCONN or MQDISC API exit function:
  - The ExitResponse2 value is ignored.

  - No further *before* exit functions in the exit chain (if any) are invoked; the API call itself is not issued.
  - For any *before* exits that were successfully called, the *after* exits are called in reverse order.
  - If registered, the termination exit functions for those *before* MQCONN or MQDISC exit functions in the chain that were successfully invoked are driven to clean up after these exit functions.
  - The MQCONN or MQDISC call fails with MQRC_API_EXIT_ERROR.
- For a *before* WebSphere MQ API exit function other than MQCONN or MQDISC:
  - The ExitResponse2 value is ignored.
  - No further *before* or *after* data conversion functions in the exit chain (if any) are invoked.
  - For any *before* exits that were successfully called, the *after* exits are called in reverse order.
  - The WebSphere MQ API call itself is not issued.
  - The WebSphere MQ API call fails with MQRC_API_EXIT_ERROR.
- For an *after* MQCONN or MQDISC API exit function:
  - The ExitResponse2 value is ignored.
  - The remaining exit functions that were successfully called before the API call are called in reverse order.
  - If registered, the termination exit functions for those *before* or *after* MQCONN or MQDISC exit functions in the chain that were successfully invoked are driven to clean up after the exit.
  - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
  - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
  - The WebSphere MQ API call is successfully issued.
- For an *after* WebSphere MQ API call exit function other than MQCONN or MQDISC:
  - The ExitResponse2 value is ignored.
  - The remaining exit functions that were successfully called before the API call are called in reverse order.
  - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
  - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
  - The WebSphere MQ API call is successfully issued.
- For the *before* data conversion on get exit function:
  - The ExitResponse2 value is ignored.
  - The remaining exit functions that were successfully called before the API call are called in reverse order.
  - The message is not converted, and the unconverted message is returned to the application.
  - A CompCode of the more severe of MQCC_WARNING and the CompCode returned by the exit is returned to the application.
  - A Reason of MQRC_API_EXIT_ERROR is returned to the application.
  - The WebSphere MQ API call is successfully issued.

  Note: As the error is with the exit, it is better to return MQRC_API_EXIT_ERROR than to return MQRC_NOT_CONVERTED.

If an exit function sets the ExitResponse2 field to a value other than one of the supported values, a value of MQXR2_DEFAULT_CONTINUATION is assumed instead.

Using and writing applications on WebSphere MQ for z/OS

WebSphere MQ for z/OS applications can be made up from programs that run in many different environments. This means that they can take advantage of the facilities available in more than one environment.

This chapter explains the WebSphere MQ facilities available to programs running in each of the supported environments. In addition,

- For information on using the WebSphere MQ-CICS bridge, see "Using and writing WebSphere MQ-CICS bridge applications for z/OS" on page 285.
- For information on using IMS and the IMS Bridge, see "IMS and IMS Bridge applications on WebSphere MQ for z/OS" on page 323.

This chapter introduces WebSphere MQ for z/OS applications, under these headings:

Important notice

Distributed queuing using CICS ISC is retained for compatibility with previous releases; there will be no further enhancements to this function. Therefore you are recommended to use the channel initiator for distributed queuing.

Environment-dependent WebSphere MQ for z/OS functions

The main differences to be considered between WebSphere MQ functions in the environments in which WebSphere MQ for z/OS runs are:

- WebSphere MQ for z/OS supplies the following trigger monitors:
  - CKTI for use in the CICS environment
  - CSQQTRMN for use in the IMS environment

  You must write your own module to start applications in other environments.
- Syncpointing using two-phase commit is supported in the CICS and IMS environments. It is also supported in the z/OS batch environment using transaction management and recoverable resource manager services (RRS). Single-phase commit is supported in the z/OS environment by WebSphere MQ itself.

- For the batch and IMS environments, the MQI provides calls to connect programs to, and to disconnect them from, a queue manager. Programs can connect to more than one queue manager.
- A CICS system can connect to only one queue manager. This can be made to happen when CICS is initiated if the subsystem name is defined in the CICS system startup job. The MQI connect and disconnect calls are tolerated, but have no effect, in the CICS environment.
- The API-crossing exit allows a program to intervene in the processing of all MQI calls. This exit is available in the CICS environment only.
- In CICS on multiprocessor systems, some performance advantage is gained because MQI calls can be executed under multiple z/OS TCBs. For more information, see the *WebSphere MQ for z/OS Concepts and Planning Guide*.

These features are summarized in Table 14.

Table 14. z/OS environmental features

	CICS	**IMS**	**Batch/TSO**
Trigger monitor supplied	Yes	Yes	No
Two-phase commit	Yes	Yes	Yes
Single-phase commit	Yes	No	Yes
Connect/disconnect MQI calls	Tolerated	Yes	Yes
API-crossing exit	Yes	No	No
Note: Two-phase commit is supported in the Batch/TSO environment using RRS.			

Program debugging facilities

WebSphere MQ for z/OS provides a trace facility that you can use to debug your programs in all environments.

Additionally, in the CICS environment you can use:

- The CICS Execution Diagnostic Facility (CEDF)
- The CICS Trace Control Transaction (CETR)
- The WebSphere MQ for z/OS API-crossing exit

On the z/OS platform, you can use any available interactive debugging tool that is supported by the programming language that you are using.

Syncpoint support

Synchronizing the start and end of units of work is necessary in a transaction processing environment so that transaction processing can be used safely.

This is fully supported by WebSphere MQ for z/OS in the CICS and IMS environments. Full support means cooperation between resource managers so that units of work can be committed or backed out in unison, under control of CICS or IMS. Examples of resource managers are DB2, CICS File Control, IMS, and WebSphere MQ for z/OS.

z/OS batch applications can use WebSphere MQ for z/OS calls to give a single-phase commit facility. This means that an application-defined set of queue operations can be committed, or backed out, without reference to other resource managers.

Two-phase commit is also supported in the z/OS batch environment using transaction management and recoverable resource manager services (RRS). For further information see "Transaction management and recoverable resource manager services" on page 187.

Recovery support

If the connection between a queue manager and a CICS or IMS system is broken during a transaction, some units of work might not be backed out successfully.

However, these units of work are resolved by the queue manager (under the control of the syncpoint manager) when its connection with the CICS or IMS system is reestablished.

The WebSphere MQ for z/OS interface with the application environment

To allow applications running in different environments to send and receive messages through a message queuing network, WebSphere MQ for z/OS provides an *adapter* for each of the environments it supports.

These adapters are the interface between application programs and WebSphere MQ for z/OS subsystems. They allow the programs to use the MQI.

The batch adapter

The *batch adapter* provides access to WebSphere MQ for z/OS resources for programs running in:

- Task (TCB) mode
- Problem or supervisor state
- Primary address space control mode

The programs must not be in cross-memory mode.

Connections between application programs and WebSphere MQ for z/OS are at the task level. The adapter provides a single connection thread from an application task control block (TCB) to WebSphere MQ for z/OS.

The adapter supports a single-phase commit protocol for changes made to resources owned by WebSphere MQ for z/OS; it does not support multiphase-commit protocols.

RRS batch adapter

The transaction management and recoverable resource manager services (RRS) adapter:

- Uses z/OS RRS for commit control.
- Supports simultaneous connections to multiple WebSphere MQ subsystems running on a single z/OS instance from a single task.
- Provides z/OS-wide coordinated commitment control (using z/OS RRS) for recoverable resources accessed through z/OS RRS compliant recoverable managers for:
  - Applications that connect to WebSphere MQ using the RRS batch adapter.
  - DB2-stored procedures executing in a DB2-stored procedures address space that is managed by a workload manager (WLM) on z/OS.

- Supports the ability to switch a WebSphere MQ batch thread between TCBs.

WebSphere MQ for z/OS provides two RRS batch adapters:

CSQBRSTB
This adapter requires you to change any MQCMIT and MQBACK statements in your WebSphere MQ application to SRRCMIT and SRRBACK respectively. (If you code MQCMIT or MQBACK in an application linked with CSQBRSTB, you receive MQRC_ENVIRONMENT_ERROR.)

CSQBRRSI
This adapter allows your WebSphere MQ application to use either MQCMIT and MQBACK or SRRCMIT and SRRBACK.

Note: CSQBRSTB and CSQBRRSI are shipped with linkage attributes AMODE(31) RMODE(ANY). If your application loads either stub below the 16 MB line, first relink the stub with RMODE(24).

Migration:

You can migrate existing Batch/TSO WebSphere MQ applications to exploit RRS coordination with few or no changes.

If you link-edit your WebSphere MQ application with the CSQBRRSI adapter, MQCMIT and MQBACK syncpoint your unit of work across WebSphere MQ and all other RRS-enabled resource managers. If you link-edit your WebSphere MQ application with the CSQBRSTB adapter, change MQCMIT and MQBACK to SRRCMIT and SRRBACK respectively. The latter approach is preferable; it clearly indicates that the syncpoint is not restricted to WebSphere MQ resources only.

The CICS adapter

If you are using the CICS adapter from a WebSphere MQ for z/OS system, ensure that CICS can obtain sufficient storage to accommodate messages up to 100 MB long.

Note to users

A CICS system can have only one connection to a WebSphere MQ for z/OS queue manager, and this connection is managed by the WebSphere MQ for z/OS *CICS adapter*. The CICS adapter provides access to WebSphere MQ for z/OS resources for CICS programs.

In addition to providing access to the MQI calls, the adapter provides:

- A trigger monitor (or task initiator) program that can start programs automatically when certain trigger conditions on a queue are met. For more information, see "Starting WebSphere MQ applications using triggers" on page 195.
- An API-crossing exit that can be invoked before and after each MQI call. For more information, see "The API-crossing exit for z/OS" on page 277.
- A trace facility to help you when debugging programs.
- Facilities that allow the MQI calls to be executed under multiple z/OS TCBs. For more information, see the *WebSphere MQ for z/OS Concepts and Planning Guide*.

CICS adapter performance considerations:

This section describes how the CICS adapter optimizes the performance of a CICS to WebSphere MQ connection.

There are a number of factors to consider when performance is critical:

First MQI call
In general, the first MQI call of a task takes longer to perform than subsequent calls. This is because the environment must be set up. For example, the adapter must acquire storage and security information, and control blocks must be allocated and formatted.

MQGET and the SIGNAL option
Using the SIGNAL option with an **MQGET** call imposes an additional overhead. This is because the SIGNAL option can produce a CICS GETMAIN in the adapter, which is used to record the address of the ECB so that it can be posted if the queue manager abends.

API-crossing exit
Using the API-crossing exit also imposes a host processor overhead on each MQI call. The overhead in handling the exit parameter block and the invocations are minimal, but the exit can be invoked twice for each MQI call through EXEC CICS LINK.

CICS tracing
CICS tracing in the adapter also increases the pathlength of an MQI call. A large number of trace entries can be generated depending on how busy the system is. There is no control over the granularity of the trace entries produced in the adapter. Therefore, tracing should only be switched on if necessary.

MQGET and the WAIT option
Using **MQGET** with the WAIT option is less efficient if the task has been put into a wait until a message arrives. The adapter implements the wait as a form of CICS wait. When a message arrives, the adapter effectively reissues the **MQGET** call for the application.

Therefore, use the WAIT option with care.

MQCLOSE
Issuing an **MQCLOSE** call is not always necessary because WebSphere MQ automatically closes any unclosed handles when the task ends.

MQPUT1
If there is only one message to be put, **MQPUT1** is more efficient than an **MQOPEN**-**MQPUT**-**MQCLOSE** sequence because only one flow is generated between the WebSphere MQ and the adapter, instead of three.

To put multiple messages, use **MQOPEN**-**MQPUT**...**MQPUT**-**MQCLOSE**.

EXEC CICS RETURN
Implicit syncpointing generated by EXEC CICS RETURN is more efficient than issuing the explicit syncpoint call EXEC CICS SYNCPOINT followed by EXEC CICS RETURN.

The EXEC CICS RETURN call accommodates all the work needed for syncpointing and task termination into one flow to WebSphere MQ instead of the two separate flows used when explicit syncpointing is used.

Two-phase commit
A two-phase commit consumes more resources than a single-phase commit, both in host processor cost and response time. This is because a two-phase commit involves one more flow to WebSphere MQ and more physical

logging. If an application is restricted to recoverable updates in WebSphere MQ and no other resource managers, CICS invokes the adapter for a single-phase commit.

Syncpoint bypassing
The adapter does not use the read-only commit feature in CICS. When a transaction is restricted to non-recoverable or non-destructive work in WebSphere MQ, syncpointing is bypassed because it is not necessary. The clean-up process is performed when the task ends.

Statistics collection
Statistics collection by connection and by task is done for each MQI call and cannot be switched off. This overhead is negligible.

You can use the CKQC transaction to display statistics for the current connection.

The adapter supports a two-phase commit protocol for changes made to resources owned by WebSphere MQ for z/OS, with CICS acting as the syncpoint coordinator.

The CICS adapter also supplies facilities (for use by system programmers and administrators) for managing the CICS-WebSphere MQ for z/OS connection, and for collecting task and connection statistics. These facilities are described in the *WebSphere MQ for z/OS System Administration Guide*.

Adapter trace points:

Application programmers can use trace points related to the MQI calls (for example, CSQCGMGD (get message data)) for debugging CICS application programs.

System programmers can use trace points related to system events, such as recovery and task switching, for diagnosing system-related problems. For full details of trace points in the CICS adapter, see the *WebSphere MQ for z/OS Problem Determination Guide*.

Some trace data addresses are passed by applications. If the address of the trace data is in the private storage area of the CICS region, the contents of the area are traced when necessary. For example, this would be done for the trace entries CSQCGMGD (get message data) or CSQCPMGD (put message data). If the address is not in the private storage area, message CSQC416I is written to the CICS trace; this contains the address in error.

Abends:

This section describes some of the things to consider with regard to CICS AEY9 and QLOP abends.

For information about all other abends, see the *WebSphere MQ for z/OS Messages and Codes*.

CICS AEY9 abends:

A transaction does *not* abend with a CICS AEY9 code if it issues an MQI call before the adapter is enabled. Instead, it receives return code MQCC_FAILED and reason code MQRC_ADAPTER_NOT_AVAILABLE.

For more information about CICS AEY9 abends, see the *CICS Messages and Codes*.

QLOP abends:

Tasks abend with the abend code QLOP if a second MQI call is made after a call has been returned with completion code MQCC_FAILED and one of these reason codes:

- MQRC_CONNECTION_BROKEN
- MQRC_Q_MGR_NAME_ERROR
- MQRC_Q_MGR_NOT_AVAILABLE
- MQRC_Q_MGR_STOPPING
- MQRC_CONNECTION_STOPPING
- MQRC_CONNECTION_NOT_AUTHORIZED

This runaway mechanism can be activated only after the adapter has been enabled once. Before the adapter has been enabled, such a task loops with reason code set to MQRC_ADAPTER_NOT_AVAILABLE. To avoid this, ensure that your applications respond to the above reason codes either by terminating abnormally or by issuing an EXEC CICS SYNCPOINT ROLLBACK and terminating normally.

If the application does not terminate at this point, it might not issue any further WebSphere MQ calls even if the connection between WebSphere MQ and CICS is reestablished. Once WebSphere MQ is reconnected to CICS, new transactions can use MQI calls as before.

Using the CICS Execution Diagnostic Facility:

You can use the CICS execution diagnostic facility (CEDF) to monitor applications that use the CICS adapter.

For details of how to use CEDF, see the *CICS Application Programming Guide*.

CEDF uses standard formatting to display MQI calls.

- Before the MQI call is executed:
  - CEDF displays the addresses of the call parameters
  - You can use the Working Storage key to verify or modify their contents
  - You can skip the call by overtyping the command with NOOP
- After the call has completed:
  - The results are returned in the program's storage
  - The return code and reason code are displayed in the call parameter list
  - You can modify them before returning to the application program

See *WebSphere MQ for z/OS Problem Determination Guide* for examples of the output produced by this facility.

The IMS adapter

If you are using the IMS adapter from a WebSphere MQ for z/OS system, ensure that IMS can obtain sufficient storage to accommodate messages up to 100 MB long.

Note to users

The *IMS adapter* provides access to WebSphere MQ for z/OS resources for:

- Online message processing programs (MPPs)
- Interactive fast path programs (IFPs)
- Batch message processing programs (BMPs)

To use these resources, the programs must be running in task (TCB) mode and problem state; they must not be in cross-memory mode or access-register mode.

The adapter provides a connection thread from an application task control block (TCB) to WebSphere MQ. The adapter supports a two-phase commit protocol for changes made to resources owned by WebSphere MQ for z/OS, with IMS acting as the syncpoint coordinator.

The adapter also provides a trigger monitor program that can start programs automatically when certain trigger conditions on a queue are met. For more information, see "Starting WebSphere MQ applications using triggers" on page 195.

If you are writing batch DL/I programs, follow the guidance given in this book for z/OS batch programs.

Writing z/OS UNIX System Services applications

The batch adapter supports queue manager connections from batch and TSO address spaces:

If we consider a Batch address space, the adapter supports connections from multiple TCBs within that address space as follows:

- Each TCB can connect to multiple queue managers using the MQCONN or MQCONNX call (but a TCB can only have one instance of a connection to a particular queue manager at any one time).
- Multiple TCBs can connect to the same queue manager (but the queue manager handle returned on any MQCONN or MQCONNX call is bound to the issuing TCB and cannot be used by any other TCB).

z/OS OpenEdition supports two types of pthread_create call:

1. Heavyweight threads, run one for each TCB, that are ATTACHed and DETACHed at thread start and end by z/OS.
2. Medium-weight threads, run one for each TCB, but the TCB can be one of a pool of long-running TCBs. The application must perform all necessary application cleanup, because, if it is connected to a server, the default thread termination that might be provided by the server at task (TCB) termination, is ***not*** always driven.

Lightweight threads are not supported. (If an application creates permanent threads that dispatch their own work requests, the ***application*** is responsible for cleaning up any resources before starting the next work request.)

WebSphere MQ for z/OS supports z/OS OpenEdition threads using the Batch Adapter as follows:

1. Heavyweight threads are fully supported as batch connections. Each thread runs in its own TCB, which is attached and detached at thread start and end. Should the thread end before issuing an MQDISC call, WebSphere MQ for z/OS performs its standard task cleanup, which includes committing any outstanding unit of work if the thread terminated normally, or backing it out if the thread terminated abnormally.

2. Medium-weight threads are fully supported, but if the TCB is going to be reused by another thread, the application must ensure that an MQDISC call, preceded by either MQCMIT or MQBACK, is issued before the next thread start. This implies that if the application has established a Program Interrupt Handler, and the application then abends, the Interrupt Handler must issue MQCMIT and MQDISC calls before reusing the TCB for another thread.

Note: Threading models do ***not*** support access to common WebSphere MQ resources from multiple threads.

The API-crossing exit for z/OS

This section contains product-sensitive programming interface information.

An exit is a point in IBM-supplied code where you can run your own code. WebSphere MQ for z/OS provides an *API-crossing exit* that you can use to intercept calls to the MQI, and to monitor or modify the function of the MQI calls. This section describes how to use the API-crossing exit, and describes the sample exit program that is supplied with WebSphere MQ for z/OS.

Note

The API-crossing exit is invoked only by the CICS adapter of WebSphere MQ for z/OS. The exit program runs in the CICS address space.

Using the API-crossing exit

You can use the API-crossing exit to:

- Operate additional security checks by examining the contents of each message before and after each MQI call
- Replace the queue name supplied in the message with another queue name
- Cancel the call and either issue a return code of 0 to simulate a successful call, or another value to indicate that the call was not performed
- Monitor the use of MQI calls in an application
- Gather statistics
- Modify input parameters on specific calls
- Modify the results of specific calls

Defining the exit program:

Before the exit can be used, an exit program load module must be available when the CICS adapter connects to WebSphere MQ for z/OS.

The exit program is a CICS program that must be named CSQCAPX and reside in a library in the DFHRPL concatenation. CSQCAPX must be defined in the CICS system definition file (CSD), and the program must be enabled.

When CSQCAPX is loaded, a confirmation message is written to the CKQC adapter control panel or to the console. If the program cannot be loaded, a diagnostic message is displayed.

How the exit is invoked:

When enabled, the API-crossing exit is invoked:

- By ***all*** applications that use the CICS adapter of WebSphere MQ for z/OS
- For the following MQI calls:
  - MQCLOSE
  - MQGET
  - MQINQ
  - MQOPEN
  - MQPUT
  - MQPUT1
  - MQSET
- Every time one of these MQI calls is made
- Both before ***and*** after a call

This means that using the API-crossing exit degrades the performance of WebSphere MQ for z/OS, so plan your use of it carefully.

The exit program can be invoked once ***before*** a call is executed, and once ***after*** the call is executed. On the before type of exit call, the exit program can modify any of the parameters on the MQI call, suppress the call completely, or allow the call to be processed. If the call is processed, the exit is invoked again after the call has completed.

Note: The exit program is not recursive. Any MQI calls made inside the exit do not invoke the exit program for a second time.

Communicating with the exit program:

After it has been invoked, the exit program is passed a parameter list in the CICS communication area pointed to by a field called DFHEICAP.

The CICS Exec Interface Block field EIBCALEN shows the length of this area. The structure of this communication area is defined in the CMQXPA assembler-language macro that is supplied with WebSphere MQ for z/OS :

```
*
MQXP_COPYPLIST       DSECT
                     DS  0D         Force doubleword alignment
MQXP_PXPB            DS   AL4       Pointer to exit parameter block
MQXP_PCOPYPARM       DS 11AL4       Copy of original plist
*
                        ORG  MQXP_PCOPYPARM
MQXP_PCOPYPARM1      DS   AL4       Copy of 1st parameter
MQXP_PCOPYPARM2      DS   AL4       Copy of 2nd parameter
MQXP_PCOPYPARM3      DS   AL4       Copy of 3rd parameter
MQXP_PCOPYPARM4      DS   AL4       Copy of 4th parameter
MQXP_PCOPYPARM5      DS   AL4       Copy of 5th parameter
MQXP_PCOPYPARM6      DS   AL4       Copy of 6th parameter
MQXP_PCOPYPARM7      DS   AL4       Copy of 7th parameter
MQXP_PCOPYPARM8      DS   AL4       Copy of 8th parameter
MQXP_PCOPYPARM9      DS   AL4       Copy of 9th parameter
MQXP_PCOPYPARM10     DS   AL4       Copy of 10th parameter
MQXP_PCOPYPARM11     DS   AL4       Copy of 11th parameter
*
MQXP_COPYPLIST_LENGTH  EQU  *-MQXP_PXPB
                       ORG  MQXP_PXPB
MQXP_COPYPLIST_AREA    DS   CL(MQXP_COPYPLIST_LENGTH)
*
```

Field *MQXP_PXPB* points to the exit parameter block, MQXP.

Field *MQXP_PCOPYPARM* is an array of addresses of the call parameters. For example, if the application issues an MQI call with parameters P1, P2, or P3, the communication area contains:

```
PXPB,PP1,PP2,PP3
```

where *P* denotes a pointer (address) and XPB is the exit parameter block.

Writing your own exit program

You can use the sample API-crossing exit program (CSQCAPX) that is supplied with WebSphere MQ for z/OS as a framework for your own program.

This is described in topic "The sample API-crossing exit program, CSQCAPX" on page 280.

When writing an exit program, to find the name of an MQI call issued by an application, examine the *ExitCommand* field of the MQXP structure. To find the number of parameters on the call, examine the *ExitParmCount* field. You can use the 16-byte *ExitUserArea* field to store the address of any dynamic storage that the application obtains. This field is retained across invocations of the exit and has the same lifetime as a CICS task.

If you are using CICS Transaction Server V3.2, you must write your exit program to be threadsafe and declare your exit program as threadsafe. If you are using earlier CICS releases, you are also recommended to write and declare your exit programs as threadsafe to be ready for migrating to CICS Transaction Server V3.2.

Your exit program can suppress execution of an MQI call by returning MQXCC_SUPPRESS_FUNCTION or MQXCC_SKIP_FUNCTION in the *ExitResponse* field. To allow the call to be executed (and the exit program to be reinvoked after the call has completed), your exit program must return MQXCC_OK.

When invoked after an MQI call, an exit program can inspect and modify the completion and reason codes set by the call.

Usage notes:

Here are some general points to consider when writing your exit program:

- For performance reasons, write your program in assembler language. If you write it in any of the other languages supported by WebSphere MQ for z/OS, you must provide your own data definition file.
- Link-edit your program as AMODE(31) and RMODE(ANY).
- To define the exit parameter block to your program, use the assembler-language macro, CMQXPA.
- Specify CONCURRENCY(THREADSAFE) when you define your exit program and any programs that your exit program calls.
- If you are using the CICS Transaction Server for OS/390 storage protection feature, your program must run in CICS execution key. That is, you must specify EXECKEY(CICS) when defining both your exit program and any programs to which it passes control. For information about CICS exit programs and the CICS storage protection facility, see the *CICS Customization Guide*.
- Your program can use all the APIs (for example, IMS, DB2, and CICS) that a CICS task-related user exit program can use. It can also use any of the MQI calls

except MQCONN, MQCONNX, and MQDISC. However, any MQI calls within the exit program do not invoke the exit program a second time.

- Your program can issue EXEC CICS SYNCPOINT or EXEC CICS SYNCPOINT ROLLBACK commands. However, these commands commit or roll back ***all*** the updates done by the task up to the point that the exit was used, and so their use is not recommended.
- Your program must end by issuing an EXEC CICS RETURN command. It must not transfer control with an XCTL command.
- Exits are written as extensions to the WebSphere MQ for z/OS code. Ensure that your exit does not disrupt any WebSphere MQ for z/OS programs or transactions that use the MQI. These are usually indicated with a prefix of CSQ or CK.
- If CSQCAPX is defined to CICS, the CICS system attempts to load the exit program when CICS connects to WebSphere MQ for z/OS. If this attempt is successful, message CSQC301I is sent to the CKQC panel or to the system console. If the load is unsuccessful (for example, if the load module does not exist in any of the libraries in the DFHRPL concatenation), message CSQC315 is sent to the CKQC panel or to the system console.
- Because the parameters in the communication area are addresses, the exit program must be defined as local to the CICS system (that is, not as a remote program).

The sample API-crossing exit program, CSQCAPX

The sample exit program is supplied as an assembler-language program. The source file (CSQCAPX) is supplied in the library **thlqual**.SCSQASMS (where **thlqual** is the high-level qualifier used by your installation). This source file includes pseudocode that describes the program logic.

The sample program contains initialization code and a layout that you can use when writing your own exit programs.

The sample shows how to:

- Set up the exit parameter block
- Address the call and exit parameter blocks
- Determine for which MQI call the exit is being invoked
- Determine whether the exit is being invoked before or after processing of the MQI call
- Put a message on a CICS temporary storage queue
- Use the macro DFHEIENT for dynamic storage acquisition to maintain reentrancy
- Use DFHEIBLK for the CICS exec interface control block
- Trap error conditions
- Return control to the caller

Design of the sample exit program:

The sample exit program writes messages to a CICS temporary storage queue (CSQ1EXIT) to show the operation of the exit.

The messages show whether the exit is being invoked before or after the MQI call. If the exit is invoked after the call, the message contains the completion code and reason code returned by the call. The sample uses named constants from the CMQXPA macro to check on the type of entry (that is, before or after the call).

The sample does not perform any monitoring function, but simply places time-stamped messages into a CICS queue indicating the type of call it is processing. This provides an indication of the performance of the MQI, as well as the proper functioning of the exit program.

Note: The sample exit program issues six EXEC CICS calls for each MQI call that is made while the program is running. If you use this exit program, WebSphere MQ for z/OS performance is degraded.

Preparing and using the API-crossing exit

The sample exit is supplied in source form only. To use the sample exit, or an exit program that you have written, create a load library, as you would for any other CICS program, as described in topic "Building CICS applications" on page 375.

- For CICS Transaction Server for OS/390 and CICS for MVS/ESA, when you update the CICS system definition (CSD) data set, the definitions you need are in the member **thlqual**.SCSQPROC(CSQ4B100).

  Note: The definitions use a suffix of MQ. If this suffix is already used in your enterprise, this must be changed before the assembly stage.

If you use the default CICS program definitions supplied, the exit program CSQCAPX is installed in a ***disabled*** state. This is because using the exit program can produce a significant reduction in performance.

To activate the API-crossing exit temporarily:

1. Issue the command CEMT S PROGRAM(CSQCAPX) ENABLED from the CICS master terminal.
2. Run the CKQC transaction, and use option 3 in the Connection pull-down to alter the status of the API-crossing exit to **Enabled**.

If you want to run WebSphere MQ for z/OS with the API-crossing exit permanently enabled, with CICS Transaction Server for OS/390 and CICS for MVS/ESA, do one of the following:

- Alter the CSQCAPX definition in member CSQ4B100, changing STATUS(DISABLED) to STATUS(ENABLED). You can update the CICS CSD definition using the CICS-supplied batch program DFHCSDUP.
- Alter the CSQCAPX definition in the CSQCAT1 group by changing the status from DISABLED to ENABLED.

In both cases you must reinstall the group. You can do this by cold-starting your CICS system or by using the CICS CEDA transaction to reinstall the group while CICS is running.

Note: Using CEDA might cause an error if any of the entries in the group are currently in use.

End of product-sensitive programming interface information.

WebSphere MQ Workflow

WebSphere MQ Workflow on z/OS is a tool that helps companies improve their business processes.

z/OS workload manager (WLM) addresses the need for:

- Managing workload distribution
- Load balancing
- Distribution of computing resources to competing workloads

WebSphere MQ support for z/OS workload manager uses a WLM-managed queue. It is recognized by a value of the INDXTYPE attribute called MSGTOKEN. The initiation queue associated with a WLM-managed queue must have TRIGTYPE defined as NONE, and no ordinary local queues must be associated with this initiation queue.

If a WebSphere MQ Workflow server application has the initiation queue open for input, WebSphere MQ updates a WLM worklist as part of commit processing of MQPUTs to the WLM-managed queue. The setting of TRIGGER or NOTRIGGER on the WLM-managed queue has no effect on the updating of this WLM worklist.

The PROCESS definition is used to provide the name of the application_environment associated with a WLM-managed queue. This is passed in the APPLICID attribute. Ensure that a WLM-managed queue uniquely references an associated process and that two processes do not specify the same APPLICID value.

Messages are retrieved from a WLM-managed queue using a unique message_token, which must be passed to MQGET. To do this, you use the message_token value (MQGMO_MSGTOKEN) and the get message match option (MQMO_MATCH_MSG_TOKEN). Workflow does not usually issue MQGET calls until the message is placed successfully on the queue. If the application needs to wait for the arrival of a message, it must set the match option to MQMO_NONE.

There are MQRC values for MQGET (MQRC_MSG_TOKEN_ERROR) and MQPUT (MQRC_MISSING_WIH and MQRC_WIH_ERROR). MQRC_MISSING_WIH is returned if a message, MQPUT to a WLM-managed queue, does not include the work information header (MQWIH). MQRC_WIH_ERROR is returned if the message data does not conform to an MQWIH. MQGET does not remove this header from the message.

Note: You might experience excessive CPU usage if your z/OS system is at Version 2.5 or earlier and the number of messages on WLM-managed queues exceeds 500.

For further information see *IBM WebSphere MQ Workflow: Concepts and Architecture*, GH12-6285 and *IBM WebSphere MQ Workflow for z/OS: Customization and Administration*, SC33-7030.

Application programming with shared queues

This section discusses some of the factors that you need to take into account when designing new applications to use shared queues, and when migrating existing applications to the shared-queue environment.

Serializing your applications

Certain types of applications might have to ensure that messages are retrieved from a queue in exactly the same order as they arrived on the queue.

For example, if WebSphere MQ is being used to shadow database updates on to a remote system, a message describing the update to a record must be processed after a message describing the insert of that record. In a local queuing environment, this is often achieved by the application that is getting the messages

opening the queue with the MQOO_INPUT_EXCLUSIVE option, thus preventing any other getting application from processing the queue at the same time.

WebSphere MQ allows applications to open shared queues exclusively in the same way. However, if the application is working from a partition of a queue (for example, all database updates are on the same queue, but those for table A have a correlation identifier of A, and those for table B a correlation identifier of B), and applications want to get messages for table A updates and table B updates concurrently, the simple mechanism of opening the queue exclusively is not possible.

If this type of application is to take advantage of the high availability of shared queues, you might decide that another instance of the application that accesses the same shared queues, running on a secondary queue manager, should take over if the primary getting application or queue manager fails.

If the primary queue manager fails, two things happen:

- Shared queue peer recovery ensures that any incomplete updates from the primary application are completed or backed out.
- The secondary application takes over processing the queue.

The secondary application might start before all the incomplete units of work have been dealt with, which could lead to the secondary application retrieving the messages out of sequence. To solve this type of problem, the application can choose to be a *serialized application*.

A serialized application uses the **MQCONNX** call to connect to the queue manager, specifying a connection tag when it connects that is unique to that application. Any units of work performed by the application are marked with the connection tag. WebSphere MQ ensures that units of work within the queue-sharing group with the same connection tag are serialized (according to the serialization options on the **MQCONNX** call).

This means that, if the primary application uses the **MQCONNX** call with a connection tag of `Database shadow retriever`, and the secondary takeover application attempts to use the **MQCONNX** call with an identical connection tag, the secondary application cannot connect to the second WebSphere MQ until any outstanding primary units of work have been completed, in this case by peer recovery.

Consider using the serialized application technique for applications that depend on the exact sequence of messages on a queue. In particular:

- Applications that must not restart after an application or queue manager failure until all commit and backout operations for the previous execution of the application are complete.

  In this case, the serialized application technique is only applicable if the application works in syncpoint.
- Applications that must not start while another instance of the same application is already running.

  In this case, the serialized application technique is only required if the application cannot open the queue for exclusive input.

Note: WebSphere MQ only guarantees to preserve the sequence of messages when certain criteria are met. These are described in the description of the **MQGET** call in the *WebSphere MQ Application Programming Reference*.

Applications that are not suitable for use with shared queues

Some features of WebSphere MQ are not supported when you are using shared queues, so applications that use these features are not suitable for the shared queue environment.

Consider the following points when designing your shared-queue applications:

- Queue indexing is limited for shared queues. If you want to use the message identifier or correlation identifier to select the message that you want to get from the queue, the queue should be indexed with the correct value. If you are selecting messages by message identifier alone, the queue needs an index type of MQIT_MSG_ID (although you can also use MQIT_NONE). If you are selecting messages by correlation identifier alone, the queue must have an index type of MQIT_CORREL_ID.
- You cannot use temporary dynamic queues as shared queues. However, you can use permanent dynamic queues. The models for shared dynamic queues have a DEFTYPE of SHAREDYN (shared dynamic) although they are created and destroyed in the same way as PERMDYN (permanent dynamic) queues.

Deciding whether to share non-application queues

There are queues other than application queues that you might want to consider sharing:

Initiation queues

If you define a shared initiation queue, you do not need to have a trigger monitor running on every queue manager in the queue-sharing group, as long as there is at least one trigger monitor running. (You can also use a shared initiation queue even if there is a trigger monitor running on each queue manager in the queue-sharing group.)

If you have a shared application queue and use the trigger type of EVERY (or a trigger type of FIRST with a small trigger interval, which behaves like a trigger type of EVERY) your initiation queue must always be a shared queue. For more information about when to use a shared initiation queue, see Table 15 on page 285.

SYSTEM.* queues

You can define the SYSTEM.ADMIN.* queues used to hold event messages as shared queues. This can be useful to check load balancing if an exception occurs. Each event message created by WebSphere MQ contains a correlation identifier indicating which queue manager produced it.

You must define the SYSTEM.QSG.* queues used for shared channels and intra-group queuing as shared queues.

You can also change the definitions of the SYSTEM.DEFAULT.LOCAL.QUEUE to be shared, or define your own default shared queue definition. This is described in the section **Defining system objects** in the *WebSphere MQ for z/OS Concepts and Planning Guide*.

You cannot define any other SYSTEM.* queues as shared queues.

Migrating your existing applications to use shared queues

Migrating your existing queues to shared queues is described in the *WebSphere MQ for z/OS System Administration Guide*.

When you migrate your existing applications, consider the following things, which might work in a different way in the shared queue environment:

Reason Codes
When you migrate your existing applications to use shared queues, check for the new reason codes that can be issued.

Triggering
If you are using a shared application queue, triggering works on committed messages only (on a non-shared application queue, triggering works on all messages).

If you use triggering to start applications, you might want to use a shared initiation queue. Table 15 describes what you need to consider when deciding which type of initiation queue to use.

Table 15. When to use a shared-initiation queue

	Non-shared application queue	**Shared application queue**
Non-shared initiation queue	As for previous releases.	If you use a trigger type of FIRST or DEPTH, you can use a non-shared initiation queue with a shared application queue. Extra trigger messages might be generated, but this setup is good for triggering long-running applications (like the CICS bridge) and provides high availability. For trigger type FIRST or DEPTH, a trigger message triggers an instance of the application on every queue manager that is running a trigger monitor and that does not already have the application queue open for input. One trigger message is generated for every queue manager; if there is more than one trigger monitor running against the non-shared local initiation queue, on a given queue manager, they will compete to process the message.
Shared initiation queue	Do not use a shared initiation queue with a non-shared application queue.	If you have a shared application queue that has a trigger type of EVERY, use a shared initiation queue or you will lose trigger messages. For trigger type FIRST or DEPTH, one trigger message is generated by each queue manager that has the named initiation queue open for input.

MQINQ
When you use the **MQINQ** call to display information about a shared queue, the values of the number of **MQOPEN** calls that have the queue open for input and output relate only to the queue manager that issued the call. No information is produced about other queue managers in the queue-sharing group that have the queue open.

Using and writing WebSphere MQ-CICS bridge applications for z/OS

Throughout this chapter the terms *CICS bridge* and *bridge* mean *WebSphere MQ-CICS bridge*.For CICS Transaction Server V3.2, the adapter and bridge are provided by CICS but the contents of this topic and its subtopics are still applicable.

Most CICS applications were developed when a 3270 terminal was the main way of interacting with users. To use these applications with new transport

mechanisms, such as WebSphere MQ, it is best to write CICS applications with the business logic of the application separated from the presentation logic. The business logic can be accessed by a CICS Distributed Program Link (DPL) request to run the program. However, not all applications can be restructured in this way, for example when the customer does not own the source code of the application. The CICS bridge protects the investment in these legacy applications by allowing them to be accessed from other platforms. This makes CICS resources readily available to programs not running under CICS. This can be done for DPL programs and for 3270 transactions.

A WebSphere MQ application can start a CICS application by sending a structured message to the CICS bridge request queue. Any data required by the CICS application can be included in the request message:

- For DPL programs, the data required is the CICS communication area (COMMAREA) data used by the application.
- For 3270 transactions, the data required is vectors describing the application data structures (ADSs) used by the application.

Similarly, the CICS application can send data back to the WebSphere MQ application in a message that is sent to a reply queue:

- For DPL programs, the data sent back is the COMMAREA data output by the application.
- For 3270 transactions, the data sent back is vectors describing the application data structures (ADSs) output by the application.

The WebSphere MQ application can run on any platform, but the bridge request queue must reside on the local z/OS queue manager that is connected to the CICS adapter.

This chapter describes how to use and design WebSphere MQ-CICS bridge applications, and contains the following sections:

- "Distributed program link applications"
- "3270 applications" on page 293
- "Information applicable to both DPL and 3270" on page 317

Distributed program link applications

This section contains the following information:

- "Using CICS DPL programs with the bridge"
- "Programming CICS DPL transactions in the distributed environment" on page 289
- "Setting fields in the MQMD and MQCIH structures (DPL)" on page 289
- "Managing MsgId and CorrelId in a unit of work (DPL)" on page 291

Using CICS DPL programs with the bridge

To link to another program that has been defined to CICS, a CICS application can issue a command like this:

```
EXEC CICS LINK PROGRAM(name) COMMAREA(data-area)
```

For the complete syntax of this command, see the *CICS Application Programming Reference*.

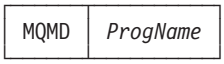
If you want a WebSphere MQ application to run a CICS application that invokes a CICS DPL program, the WebSphere MQ application must send a structured message to the bridge request queue. In the simplest case, the message data consists only of the name of a DPL program to be run. Follow this by COMMAREA data if you want to make data available to the DPL program when it starts.

If you want to run more than one DPL program within a unit of work, or you prefer a specific transaction code (overwriting the default CKBP), or you require certain levels of authorization to run the DPL program, you must supply information in an MQCIH. The MQCIH must precede the program name and any COMMAREA data that you send.

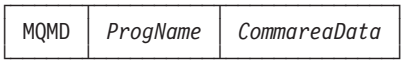
CICS DPL bridge message structure:

These examples show the different structures that you can use for messages that run DPL programs through the bridge.

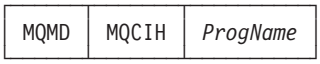
- Use this structure for an application that runs a single DPL program using default processing options, and does not send or receive COMMAREA data:

  The program specified by *ProgName* is invoked by CICS as a DPL program.
- Use this structure for an application that runs a single DPL program using default processing options, and sends and receives COMMAREA data:

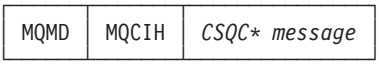
- Use this structure for an application that runs one or more DPL programs within a unit of work, or needs specific authorization to run the program, but does not send or receive COMMAREA data:

- Use this structure for an application that invokes one or more DPL programs within a unit of work, or needs specific authorization to run the program, and sends and receives COMMAREA data:

If a bridge task running a DPL program ends abnormally, it returns a message to the reply queue with the following structure, whether or not the inbound message preceding the failure contains an MQCIH:



CSQC message* represents an error message that indicates the error type. The value of field *MQCIH.Format* is set to MQFMT_STRING, so that the message can be properly converted if the final destination uses a different CCSID and encoding. The MQCIH also contains other fields that you can use to diagnose the problem.

Optionally, additional headers with format names beginning *MQH*, and containing standard link fields, can precede the MQCIH header. Such headers are returned unmodified in the output message because the bridge makes no use of data within the headers

Note:

1. The MQMD is shown in the examples to help you to visualize the overall structure of the message. This is the structure that you see if you use the general queue browser utility of WebSphere MQ SupportPac™ MA10 ″MQSeries for MVS/ESA - ISPF utilities″.
2. If you want to send only a program name, and no COMMAREA data, to the bridge, the program name must be 8 characters long. It must not be a name that is padded to the right with spaces, or the bridge reports a COMMAREA negative length error.
3. If you want to send COMMAREA data, you must pad the program name with spaces to the right, to give a total length of eight characters.
4. You can include control data in the message to specify unit of work management, and to provide data for security checking, when you include a WebSphere MQ CICS information header (CIH) in the message.

Application programming for the CICS DPL bridge:

This C-language code fragment shows how you can construct a message buffer when you want to invoke a DPL program with COMMAREA data, and include a WebSphere MQ CICS Information Header (MQCIH).

```
/*
#defines                                                    */
#define   PGMNAME "DPLPGM"              /* DPL program name  */
#define   PGMNAMELEN 8
#define   CALEN 100                     /* Commarea length   */
:
/* Data declarations                                        */
MQMD   mqmd ;                   /* Message descriptor       */
MQCIH  mqcih ;                  /* CICS information header  */
MQCHAR  * Commarea ;            /* Commarea pointer         */
MQCHAR  * MsgBuffer ;           /* Message buffer pointer   */
:
/* allocate storage for the buffers                         */

Commarea  = malloc(CALEN * sizeof(MQCHAR)) ;
MsgBuffer = malloc(sizeof(MQCIH) + PGMNAMELEN + CALEN) ;
:
/* Initialize commarea with data                            */
:
/* Initialize fields in the MQMD as required, including:    */

memcpy(mqmd.MsgId, MQMI_NONE, sizeof(mqmd.MsgId)) ;
memcpy(mqmd.CorrelId, MQCI_NEW_SESSION, sizeof(mqmd.CorrelId)) ;

/* Initialize fields in the MQCIH as required               */
:
/* Copy the MQCIH to the start of the message buffer        */

memcpy(MsgBuffer, &mqcih, sizeof(MQCIH)) ;

/* Set 8 bytes after the MQCIH to spaces                    */

memset(MsgBuffer + sizeof(MQCIH), ' ', PGMNAMELEN) ;

/* Append the program name to the MQCIH.  If it is less than */
```

```
/* 8 characters, it is now padded to the right with spaces.  */

memcpy(MsgBuffer + sizeof(MQCIH), PGMNAME, PGMNAMELEN) ;

/* Append the commarea after the program name                */

memcpy(MsgBuffer + sizeof(MQCIH) + PGMNAMELEN, &Commarea
   CALEN ) ;

/* The message buffer is now ready for the MQPUT             */
/* to the Bridge Request Queue.                              */
:
:
```

The DPL program that is invoked must conform to the DPL subset rules. See the *CICS Application Programming Guide* for further details.

Programming CICS DPL transactions in the distributed environment

CICS DPL programs and transactions can be driven through the CICS bridge when the client application resides on a workstation.

The main consideration when programming for the distributed environment is data conversion between the different encoding schemes and CCSID values of the workstation and z/OS. Conversion is carried out by two different routines, one for the MQCIH structure and another for the vector.

You can ensure that the MQCIH is converted by specifying MQFMT_CICS in the MQMD.Format field. Data conversion, however, requires a little more consideration.

If you are driving a DPL program that neither receives nor returns COMMAREA data, or if the COMMAREA data is purely character data, you can achieve data conversion by specifying MQFMT_STRING in field MQCIH.Format. If your COMMAREA data is not purely character data, you must write your own conversion routine.

Setting fields in the MQMD and MQCIH structures (DPL)

Your CICS bridge application must set a number of fields in the MQMD and the MQCIH in order to use the bridge successfully.

You need to consider the open options and the put message options that you use for the bridge request queue if the bridge monitor is started with authorization levels of VERIFY_UOW or VERIFY_ALL.

Setting the MQMD fields:

Fields in the MQMD that can affect the operation of the CICS bridge need to be initialized in your application program:

MQMD.CorrelId

For MQPUTs to the request queue, set the value to MQCI_NEW_SESSION in the first or only message in a unit of work. On subsequent messages in the unit of work, set the value to the MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your first message to the request queue.

For MQGETs from the reply queue, use the value of MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your most recent message to the request queue, or specify MQCI_NONE.

MQMD.Expiry
Set a message expiry time based on how long you want your application to wait for a reply. You are recommended to set a reasonable value for your enterprise. Set the MQCIH flags to propagate the remaining expiry time to the reply message.

MQMD.Format
Set the value to MQCICS if you include an MQCIH in the message you send to the bridge request queue; otherwise set it to the format of the data following.

MQMD.MsgId
For MQPUTs to the request queue, set MsgId to a unique value for the unit of work, or to MQMI_NONE.

For MQGETs from the reply queue, use the value of MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your first message to the request queue.

MQMD.ReplyToQ
Set the value to the name of the queue where you want the bridge to send reply messages.

The CICS bridge reply messages have the same persistence as the request messages. This means if you use persistent request messages, the reply-to queue cannot be a temporary dynamic queue or a shared queue on a non-recoverable structure.

MQMD.UserIdentifier
This field is only used when the bridge monitor is running with authorization levels of IDENTIFY, VERIFY_UOW, or VERIFY_ALL. If you use any of these, set the value to the user ID that is checked for access to the CICS DPL program.

Add the value MQOO_SET_IDENTITY_CONTEXT to the open options when you open the bridge request queue, and also add the value MQPMO_SET_IDENTITY_CONTEXT to the put message options when you send a message to the queue.

If you use this field with one of the VERIFY_* options, you must also initialize the MQCIH.Authenticator field. Set it to the value of the password or passticket associated with the user ID.

Setting the MQCIH fields:

The MQCIH contains both input and output fields; see the *WebSphere MQ Application Programming Reference* for full details of this structure. The key input fields that you need to initialize in your application program when you use the CICS DPL bridge are as follows:

MQCIH.Authenticator
This field only applies if you are using an authorization level of VERIFY_UOW or VERIFY_ALL.

Set the value to the password or passticket that is to be associated with the user ID in the MQMD.UserIdentifier field. Together, the values are used by the external security manager to determine whether the user is authorized to link to the DPL program.

If using passtickets, the *Applid* used for generating the passticket must be the same as the PASSTKTA keyword values used when starting the bridge monitor.

MQCIH.Flags
Set to MQCIH_PASS_EXPIRATION to pass the remaining expiry time to the reply message.

Set to MQCIH_REPLY_WITHOUT_NULLS to remove trailing null characters ('00'X) from the reply message.

Set to MQCIH_SYNC_ON_RETURN to specify the SYNCONRETURN option on the EXEC CICS LINK command.

You can combine the values by adding them together.

MQCIH.Format
Specifies the format of the data following the MQCIH structure. If the data is character data use MQFMT_STRING; if no conversion is needed use MQFMT_NONE.

MQCIH.GetWaitInterval
If you allow this to default, the bridge task GET WAIT interval for messages within a unit of work is the value specified on the WAIT parameter when the bridge monitor was started. If you also allow the WAIT parameter to default, the GET WAIT interval is unlimited.

MQCIH.LinkType
Specify MQCLT_PROGRAM if you are using the DPL bridge.

MQCIH.OutputDataLength
This field applies only to the DPL bridge and sets the length of data returned by the program.

MQCIH.RemoteSysId
Leave this field blank unless you need the request processed by a specific CICS system.

MQCIH.ReplyToFormat
Set this to MQFMT_NONE (the default value) if your application and the bridge are running in the same CCSID and encoding environment. Otherwise, set the value to the format of the COMMAREA data returned.

MQCIH.TransactionId
Use the default value (four spaces) unless you want the bridge to run the DPL program under a transaction code other than the default value of CKBP.

MQCIH.UOWControl
This controls the unit of work processing performed by the bridge. Allowed values are described in the*WebSphere MQ Application Programming Reference*. See also "Managing MsgId and CorrelId in a unit of work (DPL)."

Managing MsgId and CorrelId in a unit of work (DPL)

If your bridge application is running a single DPL program, set the value of *MQCIH.UOWControl* to MQCUOWC_ONLY. However, if your application is sending and receiving multiple messages, you must handle units of work correctly for the CICS DPL bridge. If you want to run multiple user programs within a unit of work, set:

- The value of *MQCIH.UOWControl* to MQCUOWC_FIRST in the first request
- MQCUOWC_MIDDLE in any intermediate requests
- MQCUOWC_LAST in the last request

Your application can send multiple request messages within a unit of work before receiving any response messages. At any time after the first message, you can terminate the unit of work by sending a message with *MQCIH.UOWControl* set to MQCUOWC_COMMIT or MQCUOWC_BACKOUT.

The following diagram summarizes the values to use and expect in key fields in the MQMD and MQCIH in typical CICS DPL bridge applications.

The first message must specify *MQMD.CorrelId* = MQCI_NEW_SESSION and subsequent messages must set *MQMD.CorrelId* to the message Id of the first message.

In Figure 20, running more than one user program using the DPL bridge, the *MsgId* of the request message is set by the queue manager (to M1), and subsequently copied to the *CorrelId*.

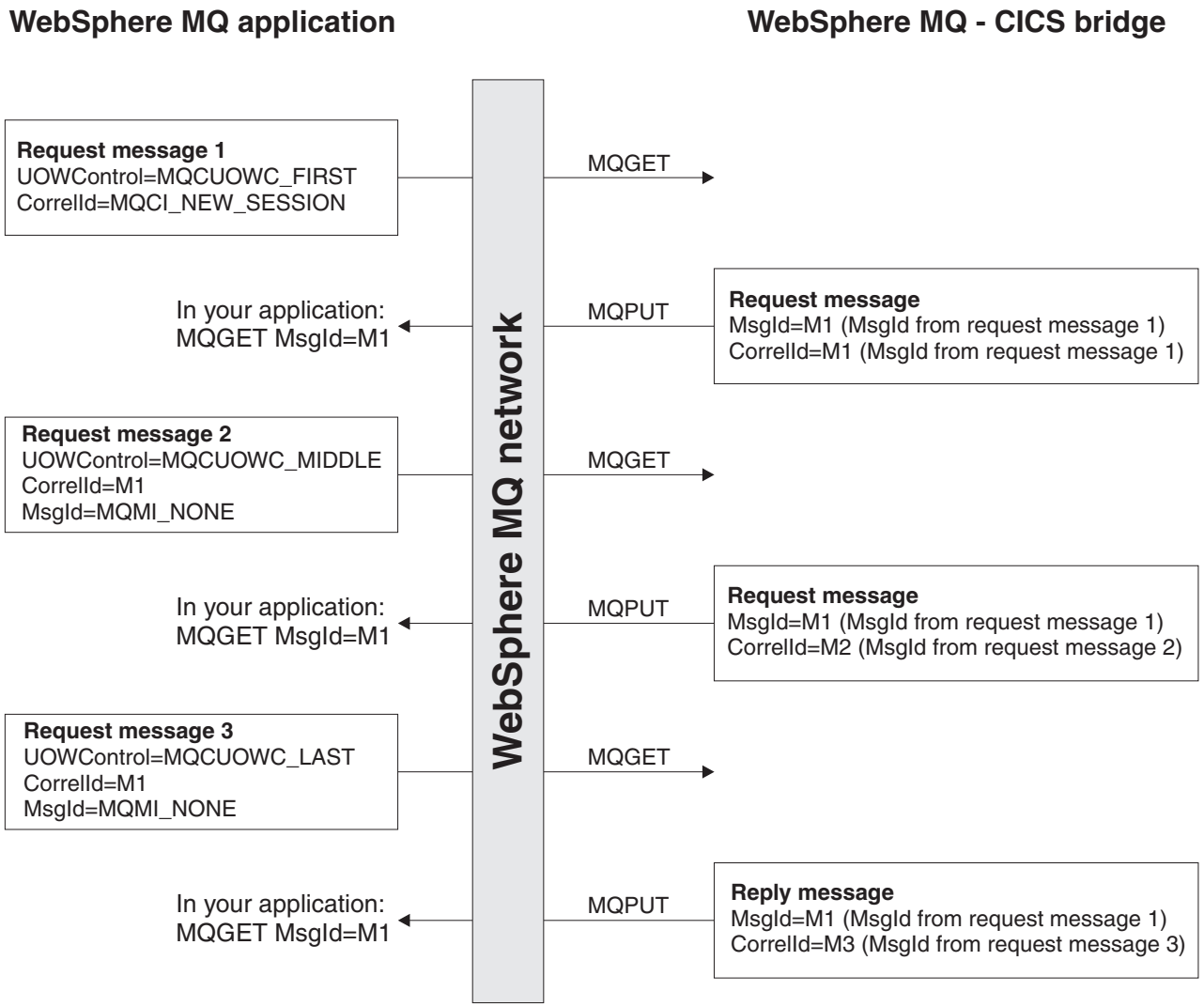


Figure 20. Setting of key fields for many CICS user programs in a unit of work viewed from the perspective of the bridge

3270 applications

This section contains the following information:

Using CICS transactions with the bridge

Without using WebSphere MQ, a CICS transaction can be started in several ways, including:

- A terminal user can enter the transaction name, followed (optionally) by data. The transaction can obtain any data that follows its identifier by issuing EXEC CICS RECEIVE when it starts.
- A preceding transaction at the terminal terminates with EXEC CICS RETURN TRANSID(*transid*); the terminal sends a 3270 data stream and starts a new transaction. A transaction that is started in this way obtains the data in the 3270 data stream by issuing EXEC CICS RECEIVE MAP or EXEC CICS RECEIVE, depending on whether it uses BMS (Basic Mapping Support) mapping or terminal control.
- An application issues an EXEC CICS START command. The started transaction issues EXEC CICS RETRIEVE to retrieve any data that has been specified on the START command.

A transaction that has been invoked at a terminal can subsequently issue commands such as EXEC CICS CONVERSE, EXEC CICS SEND MAP, and EXEC CICS RECEIVE MAP in a conversation or pseudoconversation with a terminal user.

The CICS bridge can emulate any of these ways of starting CICS transactions. It can also emulate a terminal user sending and receiving screens of data from the transaction. These emulations are achieved by using CICS bridge vectors, which represent the EXEC CICS command being emulated and provide any data that is needed. The data needed by a CICS transaction accompanies inbound messages, and the data needed by a CICS bridge application accompanies outbound messages.

Using CICS bridge vectors:

You use vectors to represent EXEC CICS commands in request and reply messages.

Vectors are identified in bridge messages by strings of numeric characters known as vector descriptors. Each vector descriptor is the CICS EIBFN value of the EXEC CICS command that it represents. For example, `0402` is the EIBFN value for EXEC CICS RECEIVE, and also the vector descriptor of the RECEIVE vector. Vectors are further qualified by the letters `I` and `O` to show whether they are inbound (to the bridge) or outbound (from the bridge).

An outbound message can contain a request vector or a reply vector. These descriptions do not mean that they go to the request queue or the reply queue; all outbound messages go to the reply queue. A CICS transaction uses a request vector to request data from the WebSphere MQ bridge application that is acting as

the virtual terminal. A CICS transaction uses a reply vector when it does not expect any data back. No such distinction is made for inbound vectors.

The following vectors are available. To get the CICS command that each represents, prefix the vector name with `EXEC CICS`.

Outbound reply vectors (no further data is required in the next inbound message):

- SEND
- SEND CONTROL
- SEND MAP
- SEND TEXT
- ISSUE ERASEAUP

Outbound request vectors (further data is required in the next inbound message):

- RECEIVE
- RECEIVE MAP
- CONVERSE

Inbound vectors:

- RECEIVE
- RECEIVE MAP
- CONVERSE
- RETRIEVE

Each of these vectors is an architected structure followed by variable length data. For details of the structures, refer to the *CICS External Interfaces Guide*.

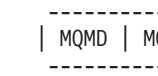
CICS 3270 bridge message structure:

The term *CICS 3270 bridge* is used here to mean all non-DPL CICS transactions.

Inbound messages:

These examples show the possible structures of CICS 3270 bridge inbound messages.

- Use this structure for an application that invokes a CICS transaction without any data:

  Set the field MQCIH.TransactionId to the name of the transaction that you want to start. Set the other fields in MQCIH to values that are appropriate for the application.
- Use this structure for inbound messages that have zero length data:

For example, an inbound RECEIVE MAP vector can represent an action where the user has only pressed a PF key. In this case, a field within the *BRMQ structure* specifies which AID key has been used, but no data follows the *BRMQ structure*.

- Use this structure for an application that invokes a transaction that will issue an EXEC CICS command that expects data to be available:

```
 ------------------------------------
| MQMD | MQCIH | BRMQ structure | data |
 ------------------------------------
```

BRMQ structure represents any of the inbound vector structures RECEIVE, RECEIVE MAP, CONVERSE, or RETRIEVE. Note that the *BRMQ structure* itself consists of a header followed by vectors and that these vectors can contain data.

Optionally, and only for CICS TS2.2 and above, additional headers with format names beginning *MQH*, and containing standard link fields, can precede the MQCIH header. Such headers are returned unmodified in the output message because the bridge makes no use of data within the headers.

- Use this structure for inbound messages that have headers before the MQCIH:

```
 ------------------------
| MQRFH2 | MQRFH2 | MQCIH |
 ------------------------
```

Outbound messages:

Outbound messages from the bridge have one of three structures, depending on whether an error occurred.

Although only a single vector is shown in each of these examples, messages can contain several concatenated vectors, except when an error occurs.

- This structure is used when bridge processing concludes normally, no errors were detected, and no data is to be returned to the bridge application:

```
 ------------------------------
| MQMD | MQCIH | BRMQ structure |
 ------------------------------
```

Even if an application abends, this is still regarded as normal completion by the bridge. The abend code issued by the application is given in the MQCIH.

- This structure is used when bridge processing concludes normally, no errors were detected, and data is to be returned to the bridge application:

```
 ------------------------------------
| MQMD | MQCIH | BRMQ structure | data |
 ------------------------------------
```

BRMQ structure represents any of the architected outbound reply or request vector structures.

- This structure is used when bridge processing concludes abnormally, an error having been detected by the bridge monitor:

```
 ----------------------------
| MQMD | MQCIH | CSQC* message |
 ----------------------------
```

CSQC message* represents an error message that indicates the error type. The value of field MQCIH.Format is set to MQFMT_STRING, to ensure that the

message can be properly converted if the final destination uses a different CCSID and encoding. The MQCIH also contains other fields that you can use to diagnose the problem.

Note:

1. The MQMD is shown in the examples to help you to visualize the overall structure of the message. This is the structure that you see if you use the general queue browser utility of WebSphere MQ SupportPac MA10 ″MQSeries for MVS/ESA - ISPF utilities″.
2. Only a single vector is shown associated with any message. In practice, a message might contain several vectors concatenated:
   - Inbound messages can contain several RECEIVE MAP vectors in anticipation of future RECEIVE MAP requests from the CICS transaction. The application needs to know the flow of control in the transaction in order to construct the message.
   - Outbound messages can contain several vectors, for example as a result of successive EXEC CICS SEND MAP commands being issued by a transaction. The CICS transaction does not control whether the outbound message contains a single vector or multiple vectors.

     If the transaction issues a command that generates a request vector, the request vector is always the last one in the message.

Application programming for the CICS 3270 bridge:

Application programming for the CICS 3270 bridge is usually more complex than application programming for the DPL bridge for these reasons:

- The bridge emulates all the functions of the CICS terminal API, including minimum function BMS.
- The bridge application needs to be aware of the internal logic and flow of control in the CICS transaction that is being run, and it must interpret and respond to vectors that it receives in outbound messages.
- If a transaction uses BMS maps, the bridge application might not have access to the copybooks created during map assembly to help interpret data in the vectors. In this case, the data must be analyzed indirectly through the use of an application data structure (ADS) descriptor.

If you are unfamiliar with the terminology used for describing application data structures, refer to the section “Application data structure terminology” on page 322.

Ensure that every inbound message that is sent to the CICS 3270 Bridge includes a vector structure after the CIH, except when you start a transaction with no data.

The vector structure definitions are available in C-language header file `dfhbrmqh.h` and COBOL copybook `DFHBRMQO`. Include these in any application that uses the bridge. These members are only provided with CICS Transaction Server on z/OS. If you want to create your application on another platform, copy them to that environment.

All the vectors have a common header, but their structures differ. Details of the structures are given in the *CICS Internet and External Interfaces Guide* for CICS V2.2, or the *CICS External Interfaces Guide* for CICS V2.3. Refer to these books when you are developing your bridge applications.

Obtain a copy of CICS SupportPac CA1E ″CICS Bridge Passthrough″ as an aid to analyzing the logic of your existing CICS transactions, and to help plan your WebSphere MQ CICS 3270 bridge applications. You might be able to use the SupportPac to test application code. It also enables you to display and capture the inbound and outbound data flows, to study how messages must be structured, and what values to insert into fields in the MQCIH and the vectors.

This example illustrates how you might write applications for the CICS 3270 bridge; it shows how to invoke a transaction that would normally be started by entering its identifier and some command line arguments at a CICS terminal:

Example: Invoking CEMT I TASK from an application:

This example shows how an application can start a transaction, in this case CEMT, that expects to receive command line arguments when it is invoked.

When the CEMT task starts, it issues EXEC CICS RECEIVE to receive any command line arguments that follow its identifier. The application that emulates the command line invocation must therefore start CEMT with a RECEIVE vector that contains appropriate values in the vector structure, and also include the command line values.

The following C-language code fragment shows how the inbound message can be constructed. Note that `dfhbrmqh.h` is in the CICS SAMPLIB.

```
/* #includes                                                  */
#include cmqc.h                 /* WebSphere MQ header       */
#include dfhbrmqh.h             /* Vector structures         */
:
:
/* #defines                                                   */
#define CMDSTRING   "CEMT I TASK"          /* Command string */
#define RCV_VECTOR  "0402"             /* Vector descriptor  */
#define INBOUND     "I   "             /* Inbound type       */
#define VERSION     "0000"             /* Vector version     */
#define YES         "Y   "             /* YES indicator      */
#define NO          "N   "             /* NO indicator       */
:
:
/* Data declarations                                          */
/* AID indicator value                                        */
const char          AID[ 4 ] = { 0x7d, ' ',' ',' ' } ;
MQMD         mqmd ;                /* Message descriptor      */
MQCIH mqcih = {MQCIH_DEFAULT} ; /* CICS information header   */
brmq_vector_header  brvh  ;     /* Standard vector header    */
brmq_receive        brrcv ;     /* RECEIVE vector structure  */
MQCHAR            * MsgBuffer ; /* Message buffer pointer    */
:
:
```

The outbound message that is returned to the reply queue contains a SEND reply vector with data in terminal control format; your application needs to know this when it analyzes the data that it receives.

Defining variables:

```
/* allocate storage for the message buffer. Note that the RECEIVE     */
/* vector structure includes space for the standard vector header.    */

MsgBuffer = malloc(sizeof(MQCIH) + sizeof(brrcv)
                + strlen(CMDSTRING) ) ;
:
:
```

Setting up the MQMD

```
memcpy(mqmd.Format, MQFMT_CICS,sizeof((MQFMT_CICS));
memcpy(mqmd.MsgId, MQMI_NONE, sizeof(MQMI_NONE));
memcpy(mqmd.CorrelId, MQCI_NEW_SESSION,     sizeof((MQCI_NEW_SESSION);
mqmd.MsgType = MQMT_REQUEST;
strncpy(mqmd.ReplyToQueue,"MyReplyQueue");
```

Setting up the MQCIH

```
mqcih.LinkType = MQCLT_TRANSACTION ;
mqcih.GetWaitInterval= 1000  ; /* one second */
mqcih.FacilityKeepTime = 10000  ; /* |= 0 says return token */
memcpy(mqcih.Facility,MQCFAC_NONE,sizeof(MQCFAC_NONE));
strncpy(mqcih.TransactionId, "CEMT", strlen("CEMT"));
strncpy(mqcih.FacilityLike, "    ", strlen("    "));
mqcih.UOWControl = MQCUOWC_FIRST;
memcpy(mqcih.AttentionId,AID,sizeof(mqcih.AttentionId); /* enter pressed */
```

Setting up the BRMQ

```
  brvh.brmq_vector_length = sizeof(brrcv) + strlen(CMDSTRING) ;
    strncpy(brvh.brmq_vector_descriptor, RCV_VECTOR,  strlen(RCV_VECTOR)) ;
    strncpy(brvh.brmq_vector_type, INBOUND, strlen(INBOUND)) ;
    strncpy(brvh.brmq_vector_version, VERSION, strlen(VERSION)) ;
    /* Initialize fields in the RECEIVE vector structure: */
    strncpy(brrcv.brmq_re_transmit_send_areas, YES, strlen(YES)) ;
    strncpy(brrcv.brmq_re_buffer_indicator, NO, strlen(NO)) ;
    strncpy(brrcv.brmq_re_aid, AID, sizeof(brrcv.brmq_re_aid)) ;
    brrcv.brmq_re_data_len =strlen(CMDSTRING) ;
```

Building the message

```
/* Copy the MQCIH to the start of the message buffer */
    memcpy(MsgBuffer, &mqcih, sizeof(MQCIH)) ;
    /* Append the RECEIVE vector to the CIH */
    memcpy(MsgBuffer + sizeof(MQCIH), brrcv, sizeof(brrcv) ) ;
    /* Overlay the standard vector header on the RECEIVE vector */
    memcpy(MsgBuffer + sizeof(MQCIH), brvh, sizeof(brvh) ) ;
    /* Append the command string to the vector structure */
    strncpy(MsgBuffer + sizeof(MQCIH) + sizeof(brrcv),
    CMDSTRING, strlen(CMDSTRING)) ;
/* the message is now ready for the MQPUT with length of
   sizeof(MQCIH) + sizeof(brrcv)+ strlen(CMDSTRING);
```

Writing applications using CICS Basic Mapping Support:

If your application does not use maps you do not need to read this section; go to "Programming CICS transactions in the distributed environment" on page 308

CICS Basic Mapping Support (BMS) provides a way for CICS applications to support a number of different terminal types. When the application issues EXEC CICS SEND MAP, BMS merges terminal-specific control data with the application data to produce a 3270 data stream that can be displayed at the terminal. When the application issues EXEC CICS RECEIVE MAP, application data is extracted from an inbound 3270 data stream and returned to the application.

A BMS map for a CICS application is created by assembling a set of BMS macros that define the characteristics of fields that are required for the display. One of the outputs from map assembly is a copybook that maps the display fields to an ADS. The CICS application must include the copybook in its data definitions so that it can address the fields in the map symbolically. The application data in a SEND MAP, and expected by a RECEIVE MAP, is mapped directly to the ADS in the copybook.

When the transaction runs under the CICS bridge, EXEC CICS SEND MAP and EXEC CICS RECEIVE MAP commands generate SEND MAP and RECEIVE MAP vectors in outbound messages. Instead of a 3270 data stream, these vectors contain ADSs equivalent to those used by the CICS application to address fields in the map.

The format of the ADS is unique for each map. It is described by a copybook created as part of map generation. Without this copybook it is difficult to interpret the data. Usually WebSphere MQ applications include the BMS copybooks so that they can create RECEIVE MAP data, and interpret SEND MAP data. However, you can write an application without the specific BMS copybooks. The format of the data is described by a structure known as the ADS descriptor (ADSD). The ADSD is added to the end of a SEND MAP vector, and it describes the format of the ADS in the vector. The ADSD contents include the names, positions, and lengths of the fields in the ADS. An ADSD can also be sent on a RECEIVE MAP request. You can use this in conversational applications to tell the WebSphere MQ application the structure of the ADS requested by the CICS application. The WebSphere MQ application can then build a RECEIVE MAP vector with this ADS, and send it as a new request.

As an application programmer, you can choose whether you want to interpret vector data in messages using the ADS, the ADSD, or the ADSDL (long form of the application data structure descriptor). In order to interpret the ADS directly, include the copybook from map assembly in your WebSphere MQ bridge application. If you want to do this, but you do not have access to the copybook or map, re-create the map with the CICS utility DFHBMSUP; this requires CICS Transaction Server 1.2 or later.

If you want to interpret the ADS indirectly through the ADSD or ADSDL, for example if you are creating a generic application that will handle all maps, you do not need to include the copybook in your bridge application. Instead you need to send control information to the bridge that tells it to include the ADSD or ADSDL in outbound SEND MAP and RECEIVE MAP request vectors as required.

If your application must run in the distributed environment, include an ADSDL in outbound SEND MAP vectors. WebSphere MQ can then convert data in the outbound message.

You can specify any of the following options by setting appropriate values in field MQCIH.ADSDescriptor in inbound messages:

- To include an ADSD (short form of the application data structure descriptor) with the SEND MAP vector, set:

  ```
  MQCIH.ADSDescriptor = MQCADSD_SEND
  ```

  If you specify this alone, you also get the short form of the ADS (application data structure) included in the SEND MAP vector.
- To include an ADSD with the RECEIVE MAP vector, set:

  ```
  MQCIH.ADSDescriptor = MQCADSD_RECV
  ```

  The ADS is never present in an outbound RECEIVE MAP request vector.
- To include an ADSDL in the SEND MAP or RECEIVE MAP vector, set:

  ```
  MQCIH.ADSDescriptor = MQCADSD_MSGFORMAT
  ```

  If you specify this, you also get the long form of the ADS included in the SEND MAP vector.

- To not include an ADS descriptor in the SEND MAP or RECEIVE MAP vector set:

  ```
  MQCIH.ADSDescriptor = MQCADSD_NONE
  ```

  This is the default. If you specify this, you will get the short form of the ADS included in the SEND MAP vector.

You can add MQCADSD_* values together to include the long form of the application data structure descriptor in both SEND MAP and RECEIVE MAP vectors:

```
MQCIH.ADSDescriptor = MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT
```

In this case, the SEND MAP vector also includes an ADS in long form.

Interpreting outbound SEND MAP and RECEIVE MAP vectors:

Include logic in your bridge application to interpret outbound SEND MAP and RECEIVE MAP request vectors, and to build and send an inbound RECEIVE MAP vector in response to the corresponding outbound RECEIVE MAP request.

SEND MAP vectors:

Include logic in your bridge application to interpret outbound SEND MAP request vectors.

An outbound SEND MAP vector can contain an application data structure (ADS) and an application data structure descriptor in short form (ADSD) or long form (ADSDL).

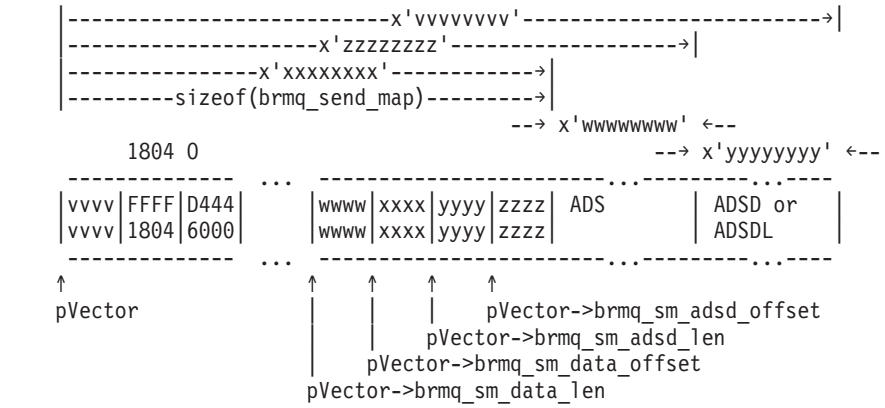
To interpret a SEND MAP vector, do the following (assuming that the message contains both an ADS and an ADSD or ADSDL):

1. Get the message containing the SEND MAP vector from the bridge reply queue into a message buffer.
2. Locate the start of the outbound SEND MAP vector in the message buffer. This is appended to the CIH, and so is at an offset equal to the length of the CIH from the start of the message buffer. You can use the following code fragment as a model:

   ```
   /* #includes                                              */
   #include cmqc.h                 /* WebSphere MQ header       */
   #include dfhbrmqh.h              /* Vector structures         */
   .
   .
   .
   /* #defines                                               */
   .
   .
   .
   MQCHAR        * MsgBuffer ;     /* Message buffer pointer    */
   brmq_send_map * pVector   ;     /* Vector pointer            */
   .
   .
   .
   /* Get message from reply queue                           */
   .
   .
   .
   /* Set the vector pointer to the start of the vector      */
   pVector = MsgBuffer + ((MQCIH *) MsgBuffer)->StrucLength ;
   ```

3. Identify the starting addresses of the application data structure (ADS) and the application data structure descriptor (ADSD or ADSDL) from the SEND MAP vector.

   The following diagram shows the structure of an outbound SEND MAP vector (assuming that you have set a pointer called *pVector* to address the start of the *brmq_send_map* vector, as in the code fragment above):



Values in the diagram shown like this:

```
ABCD
1234
```

show hexadecimal values as you would see them in an ISPF editor with *hex on*. This is equivalent to the hexadecimal value X'A1B2C3D4'.

Fields *pVector->brmq_sm_data_offset* and *pVector->brmq_sm_data_len* give the offset and length of the ADS, and fields *pVector->brmq_sm_adsd_offset* and *pVector->brmq_sm_adsd_len* give the offset and length of the ADSD or ADSDL.

Fields *brmq_sm_adsd_offset* and *brmq_sm_adsd_len* are both set to zero if no ADSD or ADSDL is included in the message.

4. Identify the fields in the ADSD or ADSDL.

   The ADSD and ADSDL are both mapped to structures that are defined in header file dfhbrarh.h, which is distributed in library <hlq>.SDFHC370 for CICS Transaction Server for OS/390 Version 1.2 or later. You can examine the structure definitions there to see how the fields are laid out. The fields of the ADSD are also described in the *CICS Internet and External Interfaces Guide* for CICS V1.2, or the *CICS External Interfaces Guide* for CICS V1.3..

   To compile your bridge application on a workstation, copy file dfhbrarh.h to that environment.

   Both the ADSD and the ADSDL are represented by two types of structure. The first structure is the descriptor, which occurs only once at the start of the ADSD or ADSDL. These types are defined as follows:

   ads_descriptor
   Descriptor for the ADSD (short form)

   ads_long_descriptor
   Descriptor for the ADSDL (long form)

   The second structure is the field descriptor, which is repeated once for each field in the map. These types are defined as follows:

   ads_field_descriptor
   Field descriptor for the ADSD (short form)

   ads_long_field_descriptor
   Field descriptor for the ADSDL (long form)

   This can be shown diagrammatically like this for the ADSDL and the ADSD:

   The ADSDL:

```
↑               ↑                 ↑
|               |                 ads_long_field_descriptor
|               ads_long_field_descriptor
ads_long_descriptor
```

The ADSD:

```
 ------------------------------------------------- ...
| ADS Descriptor | field descriptor | field descriptor |
 ------------------------------------------------- ...
↑               ↑                 ↑
|               |                 ads_field_descriptor
|               ads_field_descriptor
ads_descriptor
```

Fields *adsd_field_count* and *adsdl_field_count* in the descriptors identify the number of field descriptors in the ADSD and ADSDL.

You can use the following code fragment to set pointers to the start of the ADSD or ADSDL structures and process the field descriptors sequentially. It is assumed that *pVector* already addresses the start of the *brmq_send_map* vector, and that you have an MQCIH structure named *mqcih* that contains the CIH from the inbound message.

```
/* #includes                                                */
#include cmqc.h                /* WebSphere MQ header       */
#include dfhbrmqh.h             /* Vector structures         */
#include dfhbrarh.h             /* ADSD structures           */
:
/* Ptr to ADSD descriptor                                   */
ads_descriptor            * pADSD_D ;
/* Ptr to ADSDL descriptor                                  */
ads_long_descriptor       * pADSDL_D ;
/* Ptr to ADSD field descriptor                             */
ads_field_descriptor      * pADSD_FD ;
/* Ptr to ADSDL field descriptor                            */
ads_long_field_descriptor * pADSDL_FD ;
:
/* Initialize the pointer to the ADSDL descriptor or the    */
/* ADSD descriptor depending on mqcih.ADSDescriptor         */

if (mqcih.ADSDescriptor && MQCADSD_MSGFORMAT)
{
    pADSDL_D = pVector->brmq_sm_adsd_offset;   /* Long form  */
    pADSDL_FD = pADSDL_D + sizeof(ads_long_descriptor) ;

:
/* Enter a loop where we process all field descriptors      */
/* in the ADSDL sequentially                                 */

    do
    {
/* Perform some processing                                  */

:
pADSDL_FD += sizeof(ads_long_field_descriptor) ;
    }
    while (pADSDL_FD < pADSDL_D->adsdl_length ) ;
}

else             /* Short form                              */
{
    pADSD_D = pVector->brmq_sm_adsd_offset;    /* Short form */
    pADSD_FD = pADSD_D + sizeof(ads_descriptor) ;
/* Enter a loop where we process all field descriptors      */
/* in the ADSD sequentially                                 */
```

```
    do
    {
/* Perform some processing                                    */

.
.
.
   pADSD_FD += sizeof(ads_field_descriptor) ;
    }
    while (pADSD_FD < pADSD_D->adsd_length ) ;
}
.
.
.
```

5. Identify the fields in the ADS.

   The ADS is mapped to a structure that is generated when you assemble your map. If you include a keyword=parameter value of *DSECT*=ADSDL in your mapset definition macro, you get the long form of the ADS. The output from map assembly is a union of two structures: an input structure and an output structure. This example shows part of such a union (only the first field definition is shown for each structure, and the comments have been added following map assembly).

```
union
{
struct {
         char        dfhms1[12];        /* 12 reserved bytes         */
         int         dfhms2;            /* Offset to next field      */
         int         tranidl;           /* Data length of this field */
         int         tranidf;           /* Flag or attribute value   */
         int         dfhms3[7];         /* Extended attributes array */
         char        tranidi[4];        /* Data value of field       */
         ...
       } bmstmp1i;                      /* Input structure           */

struct {
         char        dfhms56[12];       /* 12 reserved bytes         */
         int         dfhms57;           /* Offset to next field      */
         int         dfhms58;           /* Data length of this field */
         int         tranida;           /* Flag or attribute value   */
         int         tranidc;           /* Extended attribute        */
         int         tranidp;           /* Extended attribute        */
         int         tranidh;           /* Extended attribute        */
         int         tranidv;           /* Extended attribute        */
         int         tranidu;           /* Extended attribute        */
         int         tranidm;           /* Extended attribute        */
         int         tranidt;           /* Extended attribute        */
         char        tranido[4];        /* Data value of field       */
         ...
       } bmstmp1o;                      /* Output structure          */

} bmstmp1;                              /* Union                     */
```

   The two structures are functionally identical, except that the input structure includes the extended attribute values in a seven–element array, and the output structure provides individually named fields.

   You can use the following code fragment to set pointers to the start of the ADS. The structure names shown in the example DSECT above are used for illustration. Two pointers are set, the first to address inbound data and the second to address outbound data. It is assumed that *pVector* already addresses the start of the *brmq_send_map* vector.

```
/* #includes                                                  */
#include cmqc.h                /* WebSphere MQ header       */
#include dfhbrmqh.h            /* Vector structures         */
#include  dfhbrarh.h ..        /* ADSD structures           */
#include        mydsect.h      /* DSECT from map assembly   */
```

```
.
.
.
bmstmp1i * pADSI ;            /* Pointer to the inbound ADS  */
bmstmp1o * pADSO ;            /* Pointer to the outbound ADS */
bmstmp1i * pADSI_An ;         /* Inbound ADS Anchor          */
bmstmp1o * pADSO_An ;         /* Outbound ADS Anchor         */
.
.
.
/* We are dealing with an outbound vector, so we will        */
/* initialize the outbound pointer to address the ADS        */

pADSO = pVector->brmq_sm_adsd_offset ;

/* Save initial value as anchor                              */

        pADSO_An = pADSO ;

/* Move to the start of the first field                      */

pADSO += pADSDL_FD->adsdl_field_offset ;

/* Enter a loop where we process all fields in the ADS       */
/* sequentially.  It is assumed that the value of pADSDL_FD   */
/* is being augmented to the next field descriptor in the    */
/* ADSDL with every loop. A model for this is shown in a code*/
/* fragment above.  Note that adsdl_field_offset contains    */
/* the absolute offset of the field from the start of the    */
/* ADS.                                                      */

do
{
    /* Perform some processing                               */

.
.
.
    /* Add offset of next field to ADS Anchor value          */
    /* to address the next field                             */

    pADSO = pADSO_An + pADSDL_FD->adsdl_field_offset ;
}
while (pADSDL_FD < pADSDL_D->adsd_length ) ;
.
.
.
```

The general structures of the long and short forms of the ADS are given in *CICS Transaction Server for OS/390 Version 1 Release 3: Web Support and 3270 Bridge*, an IBM Redbooks® publication.

RECEIVE MAP vectors:

A RECEIVE MAP request is a request for the client to provide a RECEIVE MAP on the next input message.

Unlike a SEND MAP vector, an outbound RECEIVE MAP request vector never contains an ADS. It contains an ADSD or ADSDL that describes the ADS data that it requires in the next inbound RECEIVE MAP vector, provided that MQCADSD_RECV has been specified in MQCIH.ADSDescriptor. The RECEIVE MAP vector structure differs from that of the SEND MAP vector. The main difference is that there are no fields giving the offset and length of the ADS.

Do the following to interpret a RECEIVE MAP vector (assuming that the message contains an ADSD or ADSDL):

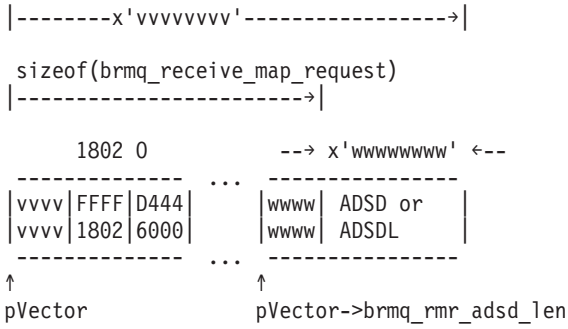
1. Get the message containing the RECEIVE MAP request vector from the bridge reply queue into a message buffer.

2. Locate the start of the outbound RECEIVE MAP vector in the message buffer. This is appended to the CIH and so is at an offset equal to the length of the CIH from the start of the message buffer. You can use this code fragment as a model.

```
/* #includes                                                  */
#include cmqc.h                 /* WebSphere MQ header       */
#include dfhbrmqh.h              /* Vector structures         */
:
/* #defines                                                   */
:
MQCHAR * MsgBuffer ;             /* Message buffer pointer    */
brmq_receive_map_request * pVector ;   /* Vector pointer      */
:
/* Get message from reply queue                               */
:
/* Set the vector pointer to the start of the vector         */
pVector = MsgBuffer + ((MQCIH *) MsgBuffer)->StrucLength ;
:
```

3. Identify the starting address ADSD or ADSDL from the RECEIVE MAP vector.

   This following diagram shows the structure of an outbound RECEIVE MAP request vector (the diagram assumes that you have set a pointer called *pVector* to address the start of the *brmq_receive_map_request* vector, as in the code fragment above).



   Values in the diagram shown like this:

```
ABCD
1234
```

   show hexadecimal values as you would see them in an ISPF editor with *hex on*. This is equivalent to the hexadecimal value X'A1B2C3D4'.

   Field *pVector->brmq_rmr_adsd_len* gives the length of the ADSD or ADSDL. No offset is given since the ADSDL is appended directly to the *brmq_receive_map_request* vector.

4. Identify the fields in the ADSD or ADSDL. To do this, proceed in general as for the SEND MAP vector described in "SEND MAP vectors" on page 300. Use the following code fragment, however, to set pointers to the start of the ADSD or ADSDL.

```
:
if (mqcih.ADSDescriptor && MQCADSD_MSGFORMAT)
{
    pADSDL_D = pVector + sizeof(brmq_receive_map_request) ;

:
}

else                                  /* Short form          */
{
```

```
    pADSD_D = pVector + sizeof(brmq_receive_map_request) ;
    .
    .
    .
    }
    .
    .
    .
```

The ADSD or ADSDL has exactly the same structure in the RECEIVE MAP vector as in the SEND MAP vector, so once you have identified its start address you can proceed as described for the SEND MAP vector.

Example of an ADSDL and an ADS:

An example showing parts of an ADSDL and an ADS is given here.

For full details of all the fields, see the references already cited. Values in the diagrams shown like this:

```
ABCD
1234
```

show hexadecimal values as you would see them in an ISPF editor with *hex on*. This is equivalent to the hexadecimal value X'A1B2C3D4'.

This diagram shows the start of the ADSDL (even though the eyecatcher shows ADSL):

```
...½ADSL..........±....CHO         L .........┌.......&$...TRANID
000BCCED0000000100040000CCD444444444D400000F000F000100054000EDCDCC44...
00581423000100OB00190003386000000000300000F000E00080000A00039159400...
↑           ↑                                  ↑   ↑       ↑
|           adsdl_field_count                   |   |       adsdl_first_field
adsdl_length                                   |   adsdl_map_columns
                                               adsdl_map_lines
```

The fields named in this example show the following:

adsdl_length
This ADSDL is 0x05B8 bytes long

adsdl_field_count
There are 0x1B (27) named fields in the ADS

adsdl_map_lines
The map has 0x18 (24) lines

adsdl_map_columns
The map has 0x50 (80) columns

adsdl_first_field
The start of the first field description in the ADSDL.

The next diagram shows the ADSDL first field descriptor and part of the next field descriptor.

```
TRANID                          ..............L ..TERMID
EDCDCC4444444444444444444444444400000000000000000D400ECDDCC4444444444...
391594000000000000000000000000000000060000000C000430003594940000000000...
↑                               ↑       ↑   ↑       ↑
adsdl_field_name                |       |   |       adsdl_next_field
                                |       |   adsdl_field_data_len
                                |       adsdl_field_offset
                                adsdl_field_name_len
```

The fields named in this example show the following:

adsdl_field_name
The name of the field in the ADS, in this case TRANID. Only the value of the field appears in the ADS, not its name.

adsdl_field_name_len
The length of the name of the field, in this case six bytes.

adsdl_field_offset
The absolute offset of the field from the start of the ADS. The offset is given as 0x0C (twelve) bytes, even though this is the first field. The reason is that the first twelve bytes of the ADS are reserved and do not contain information for the application programmer.

adsdl_field_data_len
The data length of the named field, in this case four bytes.

adsdl_next_field
The start of the next field description.

The next diagram shows the start of the ADS, which is in long form. The values here relate directly to the sample ADSDL shown above and are for the field named as TRANID in *adsdl_field_name*.

```
.....................................................BAAA............
000000000000000200000000000000000000000000000000000CCCC000200000000...
000000000000000C000000000000000000000000000000000000002111000C00000000...
↑           ↑                                       ↑   ↑
|           Offset to next field                    |   Start of next field
12 bytes reserved                                   Value of field
```

The meanings of the values shown here are as follows:

12 bytes reserved
Reserved space at the start of every ADS, in both short and long form

Offset to next field
The information given for the current field is 0x2C bytes long, from the start of this fullword length value.

Value of field
The value of the field, whose name is identified as TRANID in the ADSDL, is BAAA. The offset of the data is always 0x28 bytes from the start of the field for an ADS in long form.

Start of next field
The start of the information for the next field in the ADS.

In this case, the field information is an exact multiple of fullwords. If this were not the case, padding bytes would appear after the data value and before the next field to ensure that it started on a fullword boundary. The padding bytes would be included in the *offset to next field* value.

A number of attribute and extended attribute values for the field, not identified here, appear between the fullword giving the offset to the next field and the field value itself.

Transactions with start data:

An application that starts a transaction that will issue an EXEC CICS RETRIEVE ... QUEUE(data-area) to retrieve its start data sends a message to the bridge with a RETRIEVE vector structure, which is defined in C as *brmq_retrieve*. The structure contains a character field of length eight bytes in which the application program

must specify the name of the temporary storage queue that contains the data to be retrieved. A message containing a RETRIEVE vector is always the first in an exchange representing a conversation or pseudo conversation.

Transactions with EXEC CICS syncpoint:

Transactions that issue explicit syncpoint or rollback requests also receive an additional message on the reply queue, showing the result of the syncpoint in the MQCIH *TaskEndStatus* field. This extra message is sent with an MQMD *MsgType* of MQMT_DATAGRAM.

No input from the application is expected and the additional message is followed by the normal task end message.

Programming CICS transactions in the distributed environment

CICS DPL programs and transactions can be driven through the CICS bridge when the client application resides on a workstation.

The main consideration when programming for the distributed environment is data conversion between the different encoding schemes and CCSID values of the workstation and z/OS. Conversion is carried out by two different routines, one for the MQCIH structure and another for the vector.

You can ensure that the MQCIH is converted by specifying MQFMT_CICS in the MQMD.Format field. Vector conversion, however, requires a little more consideration.

To convert the SEND MAP and RECEIVE MAP vectors, do the following:

- Make sure that you assemble your maps specifying *DSECT*=ADSDL in your DFHMSD macro. Your map must be assembled under CICS Transaction Server for OS/390 Version 1.2 or greater for the ADSD or ADSDL to be made available. If you do not have the original mapset definition, re-create the map using the CICS DFHBMSUP utility.
- Specify a value of MQCADSD_SEND+MQCADSD_MSGFORMAT in field MQCIH.ADSDescriptor. If you are using an ADSD or ADSDL to build your RECEIVE MAP ADS, add in the value MQCADSD_RECV for this field.
- Specify a value of CSQCBDCI in field MQCIH.Format on every inbound message.
- Ensure that CONVERT=YES is specified on the channel between z/OS and the workstation.

No support is provided for conversion between workstation and mainframe formats of vectors other than SEND MAP (outbound) and RECEIVE MAP (both inbound and outbound).

From 3270 legacy to 3270 bridge - an example

This section illustrates the differences in the data flows that take place when a CICS 3270 transaction interacts with a 3270 terminal, and a CICS bridge application.

In this example, the transaction has an identifier of BAAA. It uses BMS maps, which allow the transaction to be adapted easily to run under the CICS bridge.

In the legacy environment, the transaction is started by entering its name at the CICS 3270 terminal and pressing Enter. Logic in the transaction causes it to issue

EXEC CICS SEND MAP the first time that it is invoked in a pseudoconversation, and then to terminate by issuing EXEC CICS RETURN TRANSID(BAAA).

The user enters values into fields in the map that is displayed at the terminal, and then presses an AID key. Logic in the transaction the second time that it is invoked causes it to issue EXEC CICS RECEIVE MAP to receive the map. It updates certain fields in the map by changing values in its own application data structure, and then issues EXEC CICS SEND MAP to redisplay the map at the user's terminal.

The user can then update fields in the redisplayed map, and start the RECEIVE MAP - SEND MAP cycle again. The logic can be illustrated like this (where EC represent EXEC CICS):

```
Terminal user                                  3270 Transaction
BAAA <ENTER>          ------------------->     <Initial start>
                                               <business logic>
                    <-------------------      EC SEND MAP FROM(ads)
                                               EC RETURN TRANSID(BAAA)

Update fields
<ENTER>               ------------------->     EC RECEIVE MAP INTO(ads)
                                               <business logic>
                     <-------------------      EC SEND MAP
                                               EC RETURN TRANSID(BAAA)
Update fields
<ENTER>               ------------------->     EC RECEIVE MAP
```

⋮

When the transaction runs in the bridge environment, the physical terminal is replaced by an application. The logic of the 3270 transaction is unchanged, and the application data that it receives is the same, but the data that flows, and the means by which it is transmitted, are different. Instead of a 3270 data stream, a WebSphere MQ message is used that contains an MQCIH structure (a CICS Information Header), a bridge vector structure, and optionally a representation of the application data structure.

Including these objects in the message depends on the direction in which the message flows (inbound to the bridge or outbound from the bridge), the sequence of the message in the exchange, and whether an application data structure descriptor has been requested by setting the appropriate value in a field in the MQCIH.

The section "Exact emulation - no optimization" on page 310 shows the flows that take place when the above scheme is emulated exactly. There is scope for optimization by including more than one vector in inbound messages, as shown in the section "Improved emulation, with optimization" on page 311.

It is assumed that MQCIH.ADSDescriptor is set to:

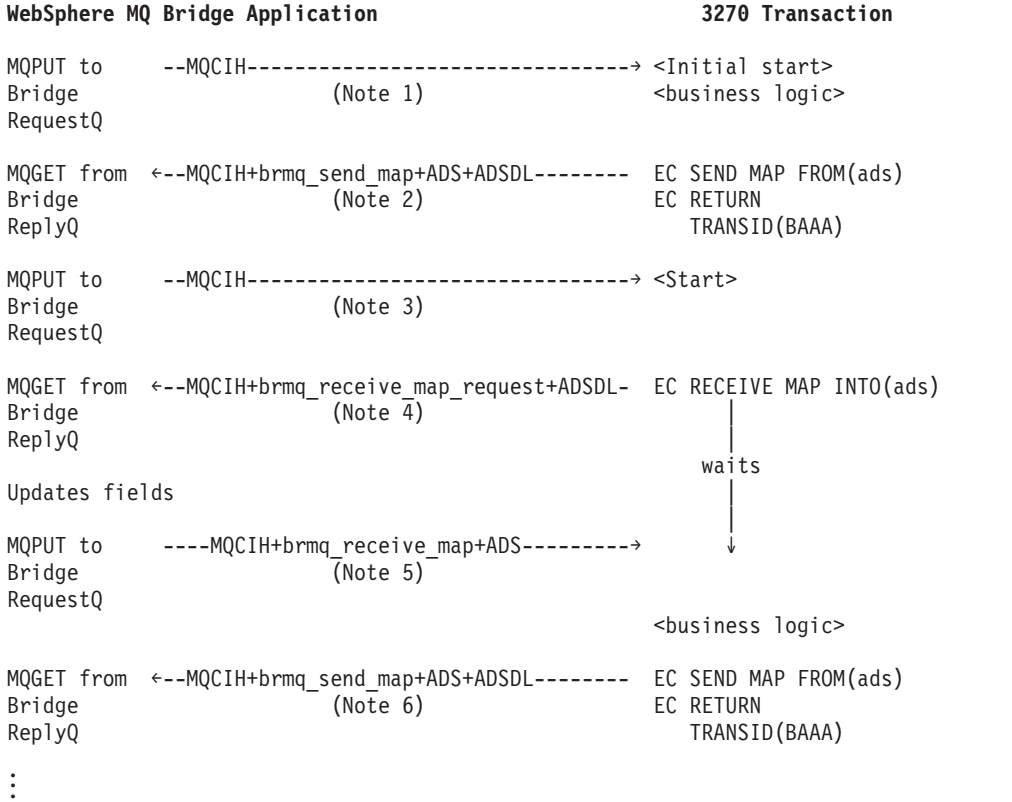
```
MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT
```

so application data structure descriptors in long form are appended to both outbound and inbound application data structures during the exchange of messages.

For clarity, the details of messaging are omitted here. For a description of the queuing model used by the CICS bridge, see the*WebSphere MQ for z/OS Concepts and Planning Guide*.

Exact emulation - no optimization:

If you examine the flows that are captured when such a transaction is run using the Passthrough tool (CICS SupportPac CA1E ″CICS 3270 Bridge Passthrough″), you can identify the structures with the help of the available online documentation.



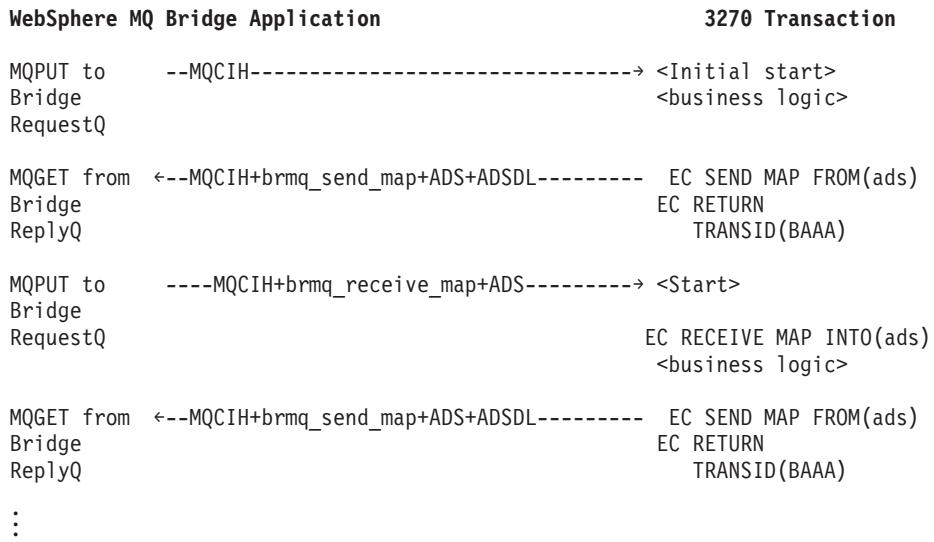
⋮

Note:

1. The initial flow from the application contains just an MQCIH. The MQCIH includes control information specifying which transaction is to be started.
2. The return flow from the 3270 transaction contains an MQCIH, which includes a facility token to be used for all subsequent flows, and diagnostic information if an error has occurred. It also contains a SEND MAP vector structure containing control information relating to the map itself, and data that represents the map. If the initiating application has requested it, an application data structure descriptor is also included.
3. The bridge application sends a message back containing only an MQCIH. This contains control information to start the transaction once more.
4. The 3270 transaction issues EXEC CICS RECEIVE MAP, as it does in the legacy environment. However, in the bridge environment the map data is not immediately available. The call is converted to a message containing an outbound RECEIVE MAP request vector. The application data structure descriptor is also included in the message. In this example, the transaction waits while the message is turned around by the bridge application. The model here is a little different from that in the legacy environment. However, the bridge architecture allows messages to contain more than one vector, so a number of requests could be satisfied by a single inbound message.
5. Having updated fields in the application data structure, the bridge application sends an inbound RECEIVE MAP reply vector to satisfy the outbound request.

6. The 3270 transaction issues EXEC CICS SEND MAP, which converts to a SEND MAP vector, and the cycle repeats.

Improved emulation, with optimization:



If you compare this sequence with the unoptimized flows shown at the section "Exact emulation - no optimization" on page 310, you can see that the CICS transaction does not have to send a RECEIVE MAP request vector, because the inbound RECEIVE MAP vector has already anticipated the requirement and the inbound map is already available to the transaction.

Setting fields in the MQMD and MQCIH structures (3270)

Your CICS bridge application must set a number of fields in the MQMD and the MQCIH in order to use the bridge successfully.

You need to consider the open options and the put message options that you use for the bridge request queue if the bridge monitor is started with authorization levels of VERIFY_UOW or VERIFY_ALL.

Setting the MQMD fields:

Fields in the MQMD that can affect the operation of the CICS bridge need to be initialized in your application program:

MQMD.CorrelId

For MQPUTs to the request queue, set the value to MQCI_NEW_SESSION in the first or only message in a unit of work. On subsequent messages in the unit of work, set the value to the MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your first message to the request queue.

For MQGETs from the reply queue, use the value of MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your most recent message to the request queue, or specify MQCI_NONE. See also "Managing MsgId and CorrelId in a unit of work (3270)" on page 314.

MQMD.Expiry

Set a message expiry time based on how long you want your application to wait for a reply. You are recommended to set a reasonable value for your enterprise. Set the MQCIH flags to propagate the remaining expiry time to the reply message.

MQMD.Format
The value must be MQFMT_CICS.

MQMD.MsgId
For MQPUTs to the request queue, set MsgId to a unique value for the unit of work, or to MQMI_NONE.

For MQGETs from the reply queue, use the value of MQMD.MsgId that WebSphere MQ set in your message descriptor when you put your first message to the request queue.

MQMD.ReplyToQ
Set the value to the name of the queue where you want the bridge to send reply messages.

The CICS bridge reply messages have the same persistence as the request messages. This means if you use persistent request messages, the reply-to queue cannot be a temporary dynamic queue or a shared queue on a non-recoverable structure.

MQMD.UserIdentifier
This field is only used when the bridge monitor is running with authorization levels of IDENTIFY, VERIFY_UOW, or VERIFY_ALL. If you use any of these, set the value to the user ID that is checked for access to the CICS resources.

Add the value MQOO_SET_IDENTITY_CONTEXT to the open options when you open the bridge request queue, and also add the value MQPMO_SET_IDENTITY_CONTEXT to the put message options when you send a message to the queue.

If you use this field with one of the VERIFY_* options, you must also initialize the MQCIH.Authenticator field. Set it to the value of the password or passticket associated with the user ID.

Setting the MQCIH fields:

The MQCIH contains both input and output fields; see the *WebSphere MQ Application Programming Reference* for full details of this structure. The key input fields that you need to initialize in your application program when you use the CICS 3270 bridge are as follows:

MQCIH.ADSDescriptor
This field applies to transactions that use BMS SEND MAP and RECEIVE MAP calls. If this is the case, and the application that is sending bridge request messages is on a workstation, set this value to MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT. This ensures that the vectors in the bridge request and reply messages are correctly converted between the different CCSID and encoding schemes of the workstation and the mainframe.

MQCIH.AttentionId
Set this field to a value representing the AID key expected by the transaction, if any; otherwise accept the default value of four spaces, which will appear to the CICS transaction as the ENTER AID key.

The inbound RECEIVE, RECEIVE MAP, and CONVERSE vectors also have fields in which you can specify AID values. The value in the MQCIH is the value to which EIBAID is set to when the application is started. It represents the PF key used to start the transaction. The value in the

inbound vector is the value used when the data is entered. For example, this would be the value of EIBAID after the EXEC CICS RECEIVE MAP instruction.

Note:

1. For conversational transactions there are separate values for the initial MQCIH value and the value on the vector.
2. If the WebSphere MQ application is sending a message in response to a request vector, the value in the MQCIH is ignored.
3. In the case of pseudoconversational transactions, enter the same value in the MQCIH and the first vector.

The first byte of this field is set to the value in the CICS copybook DFHAID.

MQCIH.Authenticator
This field only applies if you are using an authorization level of VERIFY_UOW or VERIFY_ALL.

Set the value to the password or passticket that is to be associated with the user ID in the MQMD.UserIdentifier field. Together, the values are used by the external security manager to determine whether the user is authorized to start the 3270 transaction.

If using passtickets, the *Applid* used for generating the passticket must be the same as the PASSTKTA keyword values used when starting the bridge monitor.

MQCIH.ConversationalTask
See the *WebSphere MQ Application Programming Reference* for details.

MQCIH.Facility
Set this to MQCFAC_NONE in the first message in a pseudoconversation, and also set the MQCIH.FacilityKeepTime to a non zero value. The bridge returns a facility token in the first message, and this value must be used in all subsequent inbound messages in the pseudoconversation.

MQCIH.FacilityKeepTime
If you are sending more than a single message in a pseudoconversation, set this to a non zero value (in seconds) in the first message for the bridge to return a facility token. Successive transactions in a pseudoconversation can use the same facility token once it has been set in this way, ensuring that associated terminal areas, for example the TCTUA, are preserved for the period of the pseudoconversation.

Set the value of MQCIH.Facility to MQCFAC_NONE in the first message in order to receive a facility token from the bridge.

MQCIH.FacilityLike
Either use the default value of four spaces, or specify the name of an installed terminal. You can find the names of installed terminals by entering the CICS command CEMT I TASK or a CEMT I TERM at a CICS terminal.

MQCIH.Flags
Set the value to MQCIH_PASS_EXPIRATION to pass the remaining expiry time to the reply message.

MQCIH.Format
Set the value to CSQCBDCI. This informs the bridge that any data following the MQCIH is inbound to the bridge, and might need to be

converted. The bridge sets the value of MQCIH.Format in the outbound message, which is returned to the reply queue to CSQCBDCO.

MQCIH.GetWaitInterval
If you allow this to default, the bridge task GET WAIT interval for messages within a unit of work is the value specified on the WAIT parameter when the bridge monitor was started. If you also allow the WAIT parameter to default, the GET WAIT interval is unlimited.

MQCIH.LinkType
Specify MQCLT_TRANSACTION if you are using the 3270 bridge.

MQCIH.RemoteSysid
Set to blank on the first message of a pseudo conversation unless you require the request to be processed on a specific CICS system. Set to the value returned in the first reply message in subsequent messages in the pseudo conversation.

MQCIH.StartCode
Change the value of this field from the default value of MQCSC_NONE only if you are starting a 3270 transaction. The value you use depends on the nature of the transaction. Use a value of MQCSC_START if the transaction is started by an EXEC CICS START command without data, and it does not issue EXEC CICS RETRIEVE. Use a value of MQCSC_STARTDATA if the transaction is started by an EXEC CICS START command with data, and it issues EXEC CICS RETRIEVE. Otherwise, use a value of MQCSC_TERMINPUT.

MQCIH.TransactionId
This is the transaction identifier of the user 3270 transaction to be run by the bridge task. The first message must specify the first transaction to be started. Set this field in subsequent messages to the value of MQCIH.NextTransactionId that is returned in the preceding reply message.

MQCIH.UOWControl
This controls the unit of work processing performed by the bridge. See also "Managing MsgId and CorrelId in a unit of work (3270)."

Managing MsgId and CorrelId in a unit of work (3270)

The usual style of CICS programming is pseudo-conversational, that is, a series of independent transactions that are linked together to form a complete application.

When using the 3270 bridge, the link between the transactions of a pseudo-conversation is maintained by passing the Facility token and *RemoteSysId* returned by the first transaction of the sequence into subsequent messages of the conversation

Note: In earlier versions of the CICS bridge the *RemoteSysId* field was not used; however, it is important that it is now passed thorough the conversation to enable the use of the facility for multiple CICS bridge monitors.
.

When using a CICS system prior to CICS TS 2.2, for the first message for each transaction, you must set the:

- *CorrelId* to MQCI_NEW_SESSION
- *MQCIH.UOWControl* to MQCUOWC_ONLY

Figure 21 shows a pseudo conversational 3270 transaction prior to CICS TS 2.2.

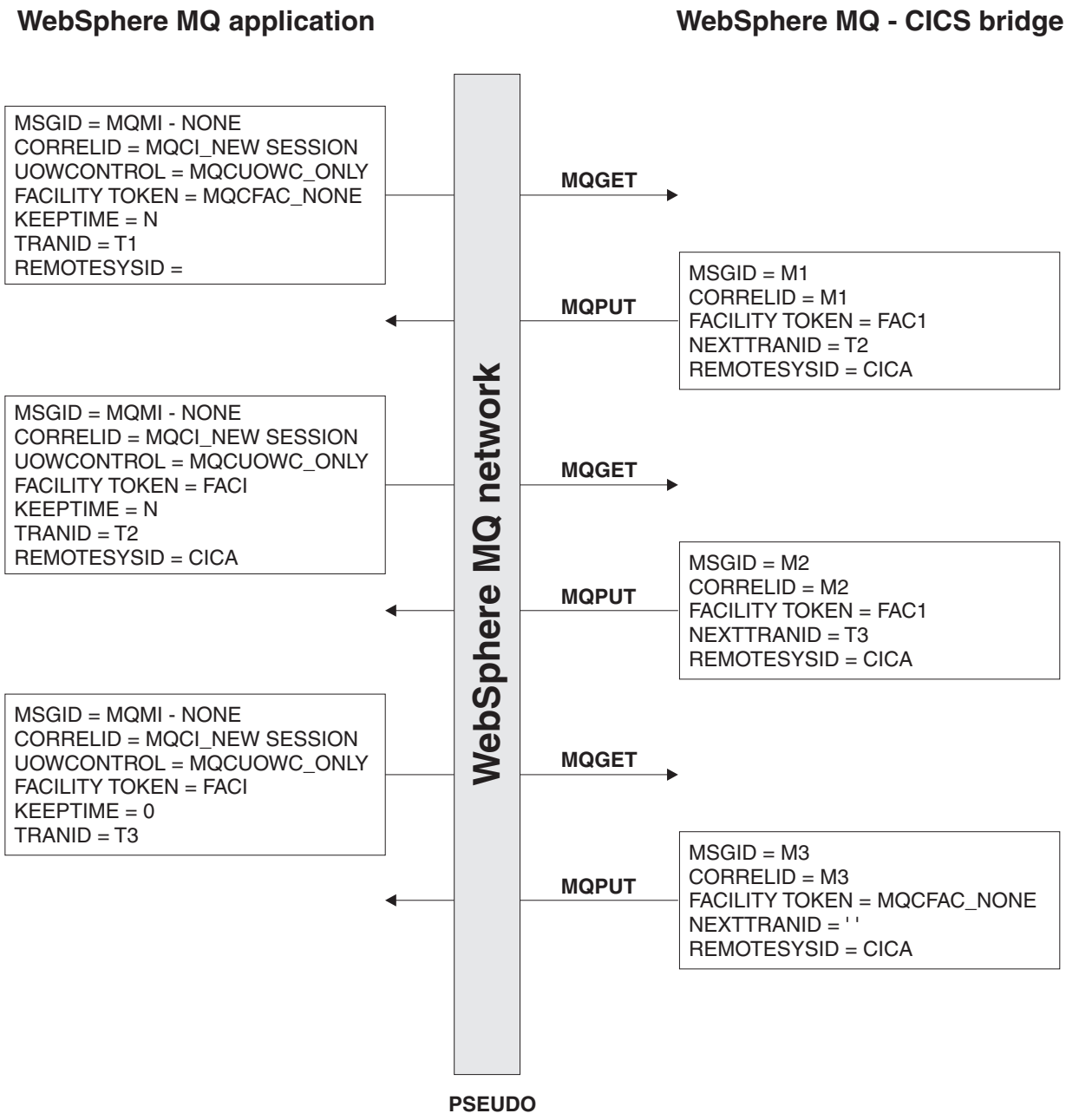


Figure 21. Setting of key fields: WebSphere MQ - pseudo-conversational 3270 transaction viewed from the perspective of the bridge prior to CICS TS 2.2

If your transaction is sending and receiving multiple messages, set:

- *MQCIH.UOWControl* to MQCUOWC_ONLY in the first message, even when a large number of messages are sent by your application
- MQCUOWC_CONTINUE in messages supplying additional data to the transaction
- *Correlid* to the message identifier generated for the first message of the transaction

If you want to end the transaction that is running, set the value of *MQCIHCancelCode* to a four-character abend code.

When using CICS TS 2.2 and subsequent releases, you can group the transactions of a pseudo-conversation together within a single bridge session, instead of using separate sessions for each transaction. Doing this reduces the overheads of the bridge monitor and improves performance. However, you can still use the multiple session approach if you need to maintain compatibility with older CICS releases, or if there might be long delays within a pseudo-conversation (for example waits for user input).

To group transactions into a single session, set:

- *MQCIH.UOWControl* to MQCUOWC_FIRST and the value of *Correlid* to MQCI_NEW_SESSION for the first message of the bridge session
- *MQCIH.UOWControl* to MQCUOWC_MIDDLE in subsequent messages of the bridge session, whether they supplying additional data to a transaction or starting a new transaction
- *MQCIH.UOWControl* to MQCUOW_LAST to indicate a proposed end of session. If the CICS transaction ends with no more requests for data, that is, the reply message type is MQMT_REPLY, the session is ended. If the CICS transaction requests more data, the *Msgtype* of the reply is MQMT_REQUEST and the next message can be sent with MQCUOWC_LAST or MQCUOWC_CONTINUE.
- If the *Msgtype* of the reply is MQMT_REQUEST and you do not want to continue the session, send *MQCIH.UOWControl* = MQCUOWC_COMMIT to end the session. To end a running 3270 transaction set *MQCIH.CancelCode* to a four-character abend code.
- *Correlid* to the message identifier generated for the first message of the transaction in all subsequent messages for the bridge session

If you want to end the transaction that is running, set the value of *MQCIHCancelCode* to a four-character abend code.

If you want to end a session between transactions set *MQCIH.UOWControl* to MQCUOWC_COMMIT.

The following diagram summarizes the values to use and expect in key fields in the MQMD and MQCIH in typical CICS 3270 bridge applications.

Figure 22 on page 317 shows a conversational 3270 transaction.

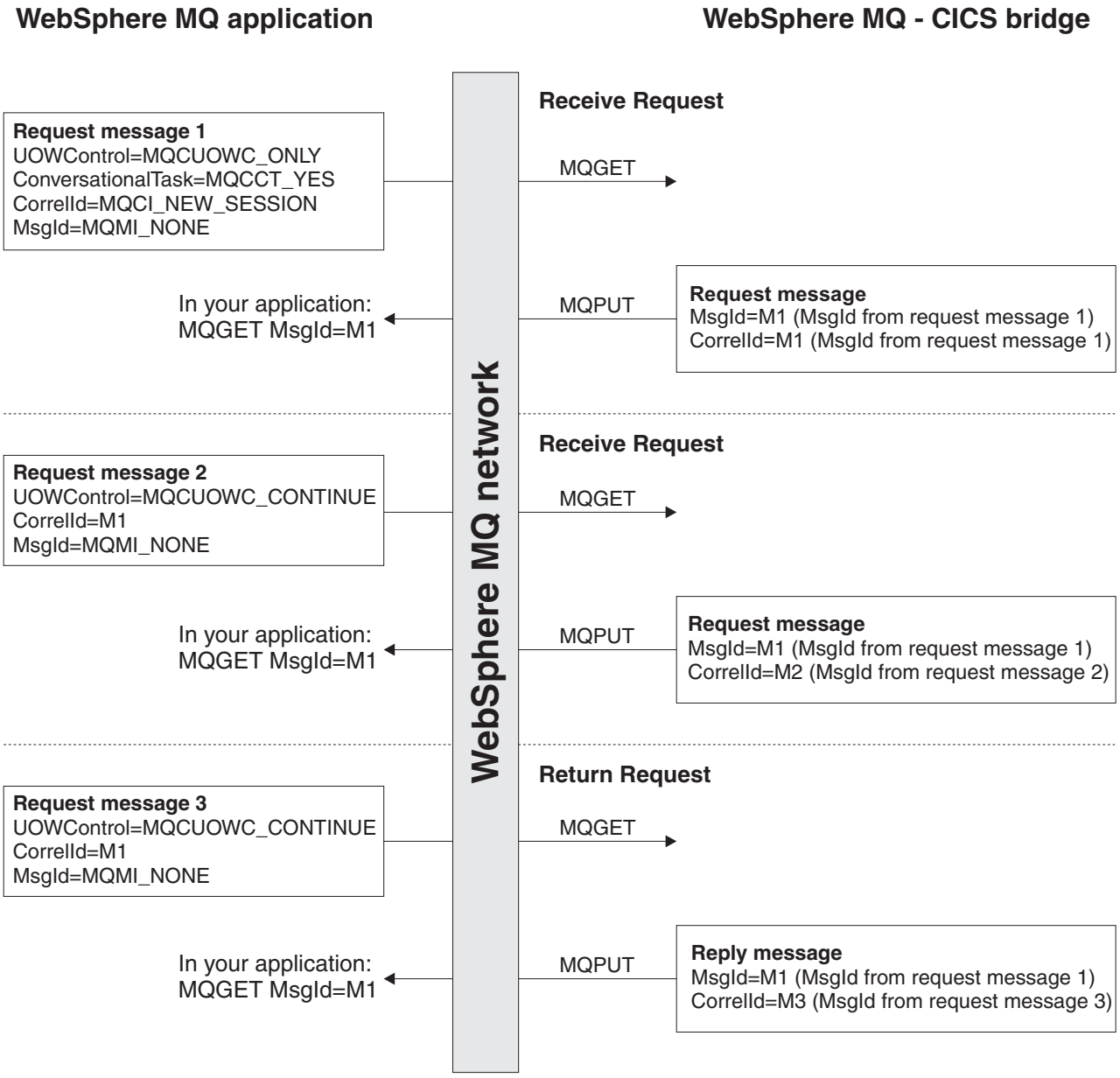


Figure 22. Setting of key fields: WebSphere MQ - conversational 3270 transaction viewed from the perspective of the bridge

Information applicable to both DPL and 3270

This section contains the following information:

Setting the open options and put message options for the bridge request queue

If you start the bridge monitor with authorization levels of IDENTIFY, VERIFY_UOW, or VERIFY_ALL, and need to control the user ID used, open the bridge request queue with open options that include MQOO_SET_IDENTITY_CONTEXT. Also include a value of MQPMO_SET_IDENTITY_CONTEXT in your put message options.

Error handling by the CICS bridge

Errors detected by the CICS bridge cause the bridge to:

- Back out the unit of work.
- If the request queue has a backout threshold (BOTHRESH) specified the request is reprocessed until it succeeds, or the backout count exceeds the threshold. You are not recommended to specify a backout threshold, unless you have reason to believe that an immediate retry is likely to be successful.
- Move request messages to the backout requeue queue, if defined, or the dead-letter queue. Messages with report option MQRO_DISCARD are not written to the dead-letter queue.
- Send an error reply message back to the client if a reply-to queue is available.
- Write a CSQC7nn message to:
  - The CICS CSMT transient data queue, or
  - The CICS joblog, or
  - Both, or
  - Issue a transaction abend.

  Where it is possible to put a message on the reply-to queue, the message contains this abend code.

Any further request messages in the same unit of work are removed from the request queue and copied to the backout requeue or dead-letter queue, either during error processing for this unit of work or at the next initialization of the monitor. No further error messages are generated.

If sending a reply message fails, the CICS bridge puts the reply on the dead-letter queue, passing identity context from the CICS bridge request queue. A unit of work is not backed out if the reply message is successfully put on the dead-letter queue. Failure to put a reply message on the dead-letter queue is treated as a request error, and the unit of work is backed out.

If the CICS bridge fails to put a request message on the dead-letter queue, the CICS bridge task abends and leaves the CICS bridge monitor to process the error. If the monitor fails to move a persistent request message to the dead-letter queue, the monitor leaves the message on the request queue, non-persistent messages are discarded .

Failure to put an error reply is ignored by the CICS bridge. The request message has already been copied to the dead-letter queue and the unit of work has been backed out by WebSphere MQ.

Figure 23 on page 319 shows what happens when an error occurs in a unit of work.

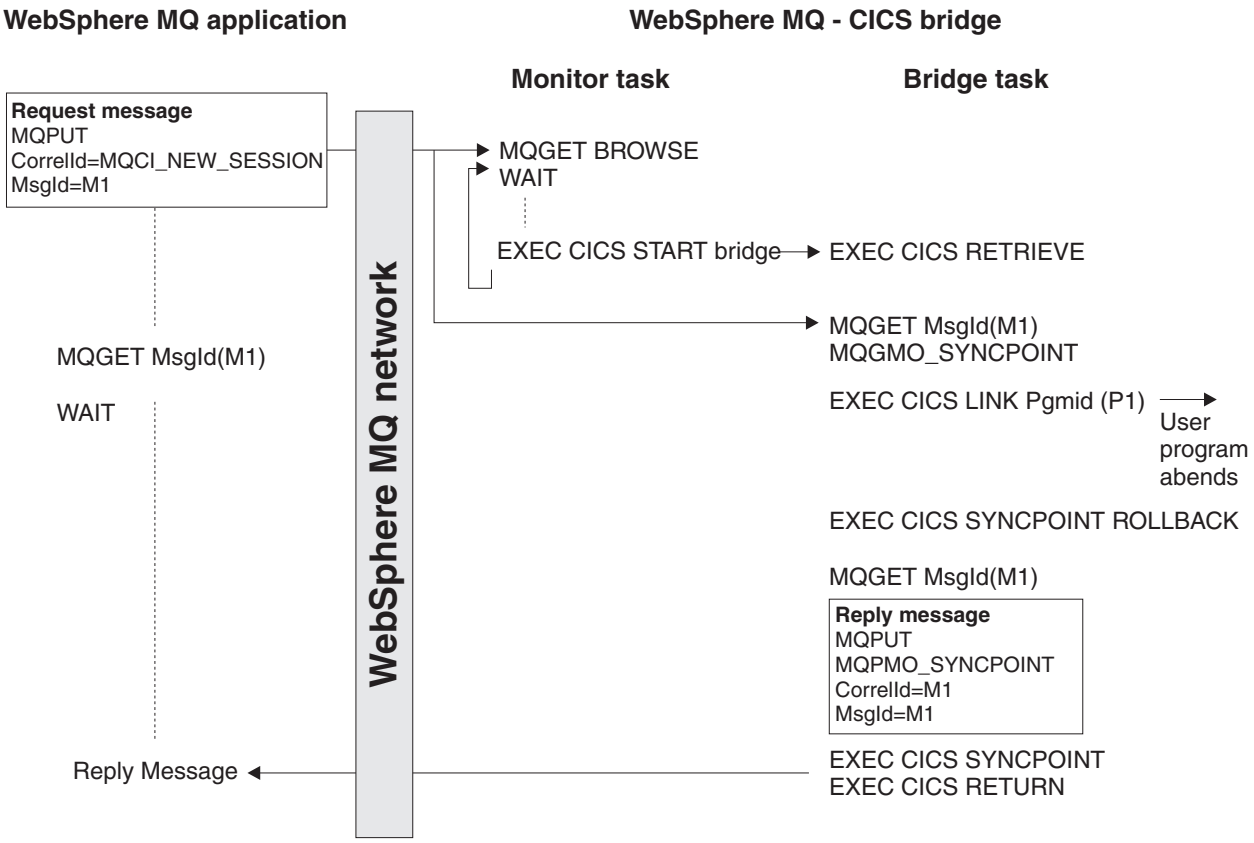


Figure 23. User program abends (only program in the unit of work)

In this figure:

- The client application sends a request message to run a CICS program named P1. The queue manager used by the client receives the message.

The monitor task browses the request queue awaiting the arrival of a message. When a message arrives, it:

- Gets the request message with browse
- Checks for any problems with the request message
- Starts a CICS bridge task
- Continues browsing the request queue

The CICS bridge task

- Gets the request message from the request queue under syncpoint control
- Takes the information in the request message and builds a COMMAREA for program P1
- Issues an EXEC CICS LINK call to program P1
- Waits for program P1 to complete

When these tasks are complete, the user program abends.

The CICS bridge task abend handler is driven, which:

- Issues an EXEC CICS SYNCPOINT ROLLBACK which:
  - Backs out all the changes made by P1
  - Reinstates the request message on the request queue

- Gets the request message a second time from the request queue, again under syncpoint control
- Copies the request to the dead-letter queue
- Puts an error reply to the reply-to queue

If the request message includes the name of a reply-to queue, the abend handler:

- Writes a CSQC7nn message to the CICS CSMT transient data queue

Debugging CICS bridge applications

This section describes some common symptoms when your bridge application appears not to work as you might expect, and suggests procedures for diagnosing the problem.

Message is PUT to the bridge request queue, but is not processed by the bridge monitor

1. Check that the bridge monitor is running. Issue CEMT I TASK and look for CKBR, or whatever other transaction identifier you are using for the bridge monitor.

   If it is not running and you are expecting it to be triggered, make sure that the triggering options on the bridge request queue are correct. Use a queue attribute of TRIGTYPE(FIRST).

   If the bridge monitor was running but is no longer running, check the output in the CICS CSMT and joblog on all CICS regions where bridge monitors should be running, to see if there has been an error that has caused the bridge monitor to terminate.
2. If the bridge request queue is defined with QSGDISP(SHARED), check that it also specifies INDXTYPE(CORRELID).
3. Browse the inbound message that is not being processed and check that the values of *MQMD.MsgId* and *MQMD.CorrelId* are correct. If this is the first message in a unit of work or a pseudoconversation, *MQMD.CorrelId* must be set to a value of MQCI_NEW_SESSION and *MQMD.MsgId* must be set to MQMI_NONE (binary zeros).
4. If this is not the first message in a unit of work or pseudoconversation, ensure that your application has checked previous reply messages adequately for possible errors. As a minimum, it should check the following fields in the MQCIH:
   - MQCIH.ReturnCode
   - MQCIH.CompCode
   - MQCIH.TaskEndStatus
   - MQCIH.AbendCode
   - MQCIH.ErrorOffset

Inbound message is taken from the request queue by the bridge monitor, but the CICS DPL program or CICS transaction fails to run

1. Check the output in the CICS MSGUSR log. This will almost always report the reason why the DPL program or transaction failed to run. The common reasons for this are:
   - Program or transaction not defined to CICS. Use CEDA to define the program or transaction and run your bridge application again.
   - Insufficient authority to run the program or transaction. Details of how to control the level of authentication used by the CICS bridge are given in *WebSphere MQ for z/OS System Setup Guide*.

2. Check the message that is sent to the reply queue by the bridge monitor. If an error has occurred, it is likely that *MQCIH.Format* is set to MQFMT_STRING and an error message is appended to the MQCIH in place of a vector.
3. Check the dead letter queue to see if a reply message has been sent there by the bridge monitor. If it has, and the values of *MQMD.MsgId* and *MQMD.CorrelId* are correct, check the value of *MQDLH.Reason*. This should be set to a feedback code that indicates the reason for the failure.

   For information on feedback codes, including those specific to the CICS Bridge, see "MQMD - Message descriptor" in the *WebSphere MQ Application Programming Reference*.

Bridge task abends

Abend codes are set in outbound messages in field MQCIH.AbendCode. In addition, the output in the CICS MSGUSR log reports abend codes for failing bridge tasks.

Abends ABR*, ABS* and ABX* are CICS bridge abends, and are documented in *CICS Messages and Codes*. Abends MBR* and MQB* are WebSphere MQ bridge abends, and are documented in *WebSphere MQ for z/OS Messages and Codes*.

Some common abend codes can be dealt with as follows:

ABRG An invalid bridge facility token was specified in an inbound message. Your first inbound message must always specify a value of MQCFAC_NONE in field *MQCIH.Facility*, and a non zero value in *MQCIH.FacilityKeepTime*. CICS returns a facility token in field *MQCIH.Facility*, and you can use this value in all subsequent inbound messages in the pseudoconversation.

ABXH Caused either by having *brmq_re_buffer_indicator* set to N, when a receive with the buffer option was specified, or having *brmq_re_buffer_indicator* set to Y and a receive (without the buffer option) specified.

MBRJ The MQCIH has invalid data. Check the values in the MQCIH field by field to find the one that is out of range. MBRJ can also be caused by a length mismatch, for example, when the *brmq_vector_length* and the length of the data vector do not agree, or there is not enough data in the CICS headers and vector

MBRN
The message is shorter than expected. There are one or two data length fields in every vector structure. The first is the first fullword field in the standard header for all vectors, and it should be equal to the overall length of the vector including the variable length data. Some vectors also contain another fullword length field that gives just the length of the variable length data. If these values indicate more data than there actually is, the bridge task will abend MBRN.

MBRO and MBRP
There is an error in the vector structure (not the variable length data). The MQCIH field ERROROFFSET gives the offset of the field in error. Check the values of the fields in the vector against the permitted values, which are described in the *CICS Internet and External Interfaces Guide* for CICS V1.2, or the *CICS External Interfaces Guide* for CICS V1.3..

Bridge monitor errors

Some errors can cause the bridge monitor transaction, CKBR, to terminate unexpectedly. If you are using triggered queues to start the monitor, and there are still messages on the bridge request queue, the CKTI transaction might attempt to restart CKBR. If the original error persists, this can lead to a loop of CKBR failures. To halt the loop, set off the *TriggerControl* attribute of the request queue while you diagnose and fix the underlying problem.

The bridge monitor can fail if it does not have sufficient authority to access the queues or CICS transactions, if it cannot write to the dead letter queue or it encounters problems when executing CICS or MQ services.

Application data structure terminology

An explanation of application data structure and application data structure descriptor terminology and their respective mnemonics.

Application data structure

The application data structure is the copybook generated while assembling a BMS map. It has the mnemonic ADS, and it can be created in short form or long form. You sometimes see the long form referred to as ADSL.

The short form of the ADS has fields that are not fullword aligned, whereas the long form of the ADS has all its fields fullword aligned.

The short form of the ADS is generated by default by map assembly. To obtain the long form of the ADS, assemble your BMS maps with the following parameters specified in the DFHMSD macro:

```
MSETNAM   DFHMSD
⋮
DSECT=ADSL,        *
LANG=C,            *
⋮
```

If you examine the DSECT that is produced on map assembly, you will see that all of the fields are fullword aligned. Be aware that this significantly increases the size of the application data structure and any message that includes it.

This option is only available for programs written in the C language. However, you can create COBOL data structures from such a DSECT by manually creating them. *CICS Transaction Server for OS/390 Version 1 Release 3: Web Support and 3270 Bridge*, an IBM Redbooks publication, gives examples of this.

Application data structure descriptor

The application data structure descriptor is an architected structure that allows an application to interpret the application data structure in a vector without having access to the copybook generated during map assembly.

There are two forms of the application data structure descriptor:

- The application data structure descriptor in short form contains fields that are not fullword aligned. It has the mnemonic ADSD.
- The application data structure descriptor in long form contains fields that are all fullword aligned. It has the mnemonic ADSDL.

ADSL – an ambiguous mnemonic

The mnemonic ADSL is ambiguous, and is sometimes used to refer to the application data *structure* in long form, and sometimes the application data structure *descriptor* in long form. The correct mnemonic for the application

data structure descriptor, long form, is ADSDL. However, if you look at the eye-catcher in the ADSDL you will see that it is ADSL, which leads to the ambiguity. The correct use of ADSL is to describe the application data structure, not its descriptor, in long form.

IMS and IMS Bridge applications on WebSphere MQ for z/OS

This chapter helps you to write IMS applications using WebSphere MQ.

- To use syncpoints and MQI calls in IMS applications, see "Writing IMS applications using WebSphere MQ."
- To write applications that exploit the WebSphere MQ-IMS Bridge, see "Writing WebSphere MQ-IMS bridge applications" on page 327.

Writing IMS applications using WebSphere MQ

This section discusses the following subjects that you should consider when using WebSphere MQ in IMS applications:

- "Syncpoints in IMS applications"
- "MQI calls in IMS applications"

Syncpoints in IMS applications

In an IMS application, you establish a syncpoint by using IMS calls such as GU (get unique) to the IOPCB and CHKP (checkpoint). To back out all changes since the previous checkpoint, you can use the IMS ROLB (rollback) call. For more information, see the following:

- *IMS/ESA Application Programming: Transaction Manager*
- *IMS/ESA Application Programming: Design Guide*

The queue manager is a participant in a two-phase commit protocol; the IMS syncpoint manager is the coordinator. Other syncpoint coordinators are not supported by the IMS adapter.

All open handles are closed by the IMS adapter at a syncpoint (except in a batch-oriented BMP). This is because a different user could initiate the next unit of work and WebSphere MQ security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made.

Handles are also closed after a ROLB call unless you are running IMS Version 3 or a batch-oriented BMP.

If an IMS application (either a BMP or an MPP) issues the MQDISC call, open queues are closed but no implicit syncpoint is taken. If the application ends normally, any open queues are closed and an implicit commit occurs. If the application ends abnormally, any open queues are closed and an implicit backout occurs.

MQI calls in IMS applications

This section covers the use of MQI calls in the following types of IMS applications:

- "Server applications" on page 324
- "Enquiry applications" on page 326

Server applications:

Here is an outline of the MQI server application model:

```
 Initialize/Connect
  .
 Open queue for input shared
  .
 Get message from WebSphere MQ queue
  .
 Do while Get does not fail
       .
     If expected message received
       Process the message
     Else
       Process unexpected message
     End if
       .
     Commit
       .
     Get next message from WebSphere MQ queue
       .
 End do
 .
 Close queue/Disconnect
 .
END
```

Sample program CSQ4ICB3 shows the implementation, in C/370™, of a BMP using this model. The program establishes communication with IMS first, and then with WebSphere MQ:

```
main()
----
    Call InitIMS
    If IMS initialization successful
       Call InitMQM
       If WebSphere MQ initialization successful
          Call ProcessRequests
          Call EndMQM
       End-if
    End-if

Return
```

The IMS initialization determines whether the program has been called as a message-driven or a batch-oriented BMP and controls WebSphere MQ queue manager connection and queue handles accordingly:

```
InitIMS
-------
Get the IO, Alternate and Database PCBs
Set MessageOriented to true

Call ctdli to handle status codes rather than abend
If call is successful (status code is zero)
   While status code is zero
      Call ctdli to get next message from IMS message queue
      If message received
         Do nothing
         Else if no IOPBC
            Set MessageOriented to false
            Initialize error message
            Build 'Started as batch oriented BMP' message
            Call ReportCallError to output the message
         End-if
         Else if response is not 'no message available'
```

```
            Initialize error message
            Build 'GU failed' message
                Call ReportCallError to output the message
                Set return code to error
         End-if
      End-if
   End-while
Else
   Initialize error message
   Build 'INIT failed' message
   Call ReportCallError to output the message
   Set return code to error
End-if

Return to calling function
```

The WebSphere MQ initialization connects to the queue manager and opens the queues. In a message-driven BMP this is called after each IMS syncpoint is taken; in a batch-oriented BMP, this is called only during program startup:

```
InitMQM
-------
    Connect to the queue manager
    If connect is successful
       Initialize variables for the open call
       Open the request queue
       If open is not successful
          Initialize error message
          Build 'open failed' message
          Call ReportCallError to output the message
          Set return code to error
       End-if
    Else
       Initialize error message
       Build 'connect failed' message
       Call ReportCallError to output the message
       Set return code to error
    End-if

    Return to calling function
```

The implementation of the server model in an MPP is influenced by the fact that the MPP processes a single unit of work per invocation. This is because, when a syncpoint (GU) is taken, the connection and queue handles are closed and the next IMS message is delivered. This limitation can be partially overcome by one of the following:

- **Processing many messages within a single unit-of-work**

  This involves:

  – Reading a message
  – Processing the required updates
  – Putting the reply

  in a loop until all messages have been processed or until a set maximum number of messages has been processed, at which time a syncpoint is taken.

  Only certain types of application (for example, a simple database update or inquiry) can be approached in this way. Although the MQI reply messages can be put with the authority of the originator of the MQI message being handled, the security implications of any IMS resource updates need to be addressed carefully.

- **Processing one message per invocation of the MPP and ensuring multiple scheduling of the MPP to process all available messages.**

Use the WebSphere MQ IMS trigger monitor program (CSQQTRMN) to schedule the MPP transaction when there are messages on the WebSphere MQ queue and no applications serving it.

If trigger monitor starts the MPP, the queue manager name and queue name are passed to the program, as shown in the following COBOL code extract:

```
* Data definition extract
 01  WS-INPUT-MSG.
     05 IN-LL1                  PIC S9(3) COMP.
     05 IN-ZZ1                  PIC S9(3) COMP.
     05 WS-STRINGPARM           PIC X(1000).
 01  TRIGGER-MESSAGE.
     COPY CMQTMC2L.
*
* Code extract
 GU-IOPCB SECTION.
     MOVE SPACES TO WS-STRINGPARM.
     CALL 'CBLTDLI' USING GU,
                          IOPCB,
                          WS-INPUT-MSG.
     IF IOPCB-STATUS = SPACES
         MOVE WS-STRINGPARM TO MQTMC.
*    ELSE handle error
*
* Now use the queue manager and queue names passed
     DISPLAY 'MQTMC-QMGRNAME          ='
             MQTMC-QMGRNAME OF MQTMC '='.
     DISPLAY 'MQTMC-QNAME             ='
             MQTMC-QNAME    OF MQTMC '='.
```

The server model, which is expected to be a long running task, is better supported in a batch processing region, although the BMP cannot be triggered using CSQQTRMN.

Enquiry applications:

A typical WebSphere MQ application initiating an inquiry or update works as follows:

- Gather data from the user
- Put one or more WebSphere MQ messages
- Get the reply messages (you might have to wait for them)
- Provide a response to the user

Because messages put on to WebSphere MQ queues do not become available to other WebSphere MQ applications until they are committed, they must either be put out of syncpoint, or the IMS application must be split into two transactions.

If the inquiry involves putting a single message, you can use the *no syncpoint* option; however, if the inquiry is more complex, or resource updates are involved, you might get consistency problems if failure occurs and you do not use syncpointing.

To overcome this, you can split IMS MPP transactions using MQI calls using a program-to-program message switch; see *IMS/ESA Application Programming: Data Communication* for information about this. This allows an inquiry program to be implemented in an MPP:

```
Initialize first program/Connect
 .
Open queue for output
 .
Put inquiry to WebSphere MQ queue
```

```
  .
 Switch to second WebSphere MQ program, passing necessary data in save
 pack area (this commits the put)
  .
END
  .
  .
 Initialize second program/Connect
  .
 Open queue for input shared
  .
 Get results of inquiry from WebSphere MQ queue
  .
 Return results to originator
 .
END
```

Writing WebSphere MQ-IMS bridge applications

This section discusses writing applications to exploit the WebSphere MQ-IMS bridge.

The following topics are discussed:

- "How the WebSphere MQ-IMS bridge deals with messages"
- "Writing your program" on page 334
- "Triggering" on page 335

For information about the WebSphere MQ-IMS bridge, see the *WebSphere MQ for z/OS Concepts and Planning Guide*.

How the WebSphere MQ-IMS bridge deals with messages

When you use the WebSphere MQ-IMS bridge to send messages to an IMS application, you need to construct your messages in a special format.

You must also put your messages on WebSphere MQ queues that have been defined with a storage class that specifies the XCF group and member name of the target IMS system. These are known as MQ-IMS bridge queues, or simply **bridge** queues.

A user does not need to sign on to IMS before sending messages to an IMS application. The user ID in the *UserIdentifier* field of the MQMD structure is used for security checking. The level of checking is determined when WebSphere MQ connects to IMS, and is described in the security section of the *WebSphere MQ for z/OS System Setup Guide*. This enables a pseudo signon to be implemented.

The WebSphere MQ-IMS bridge accepts the following types of message:

- Messages containing IMS transaction data and an MQIIH structure (described in the WebSphere MQ Application Programming Reference):

  ```
  MQIIH LLZZ<trancode><data>[LLZZ<data>][LLZZ<data>]
  ```

  Note:

  1. The square brackets, [ ], represent optional multi-segments.
  2. Set the *Format* field of the MQMD structure to MQFMT_IMS to use the MQIIH structure.

- Messages containing IMS transaction data but no MQIIH structure:

  ```
  LLZZ<trancode><data>  \
  [LLZZ<data>][LLZZ<data>]
  ```

WebSphere MQ validates the message data to ensure that the sum of the LL bytes plus the length of the MQIIH (if it is present) is equal to the message length.

When the WebSphere MQ-IMS bridge gets messages from the bridge queues, it processes them as follows:

- If the message contains an MQIIH structure, the bridge verifies the MQIIH (see the *WebSphere MQ Application Programming Reference*), builds the OTMA headers, and sends the message to IMS. The transaction code is specified in the input message. If this is an LTERM, IMS replies with a DFS1288E message. If the transaction code represents a command, IMS executes the command; otherwise the message is queued in IMS for the transaction.
- If the message contains IMS transaction data, but no MQIIH structure, the IMS bridge makes the following assumptions:
  - The transaction code is in bytes 5 through 12 of the user data
  - The transaction is in nonconversational mode
  - The transaction is in commit mode 0 (commit-then-send)
  - The *Format* in the MQMD is used as the *MFSMapName* (on input)
  - The security mode is MQISS_CHECK

  The reply message is also built without an MQIIH structure, taking the *Format* for the MQMD from the *MFSMapName* of the IMS output.

The WebSphere MQ-IMS bridge uses one or two Tpipes for each WebSphere MQ queue:

- A synchronized Tpipe is used for all messages using Commit mode 0 (COMMIT_THEN_SEND) (these show with SYN in the status field of the IMS /DIS TMEMBER client TPIPE xxxx command)
- A non-synchronized Tpipe is used for all messages using Commit mode 1 (SEND_THEN_COMMIT)

The Tpipes are created by WebSphere MQ when they are first used. A non-synchronized Tpipe exists until IMS is restarted. Synchronized Tpipes exist until IMS is cold started. You cannot delete these Tpipes yourself.

Mapping WebSphere MQ messages to IMS transaction types:

Table 16. Mapping WebSphere MQ messages to IMS transaction types

WebSphere MQ message type	Commit-then-send (mode 0) - uses synchronized IMS Tpipes	Send-then-commit (mode 1) - uses non-synchronized IMS Tpipes
Persistent WebSphere MQ messages	• Recoverable full function transactions • Irrecoverable transactions are rejected by IMS	• Fastpath transactions • Conversational transactions • Full function transactions
Nonpersistent WebSphere MQ messages	• Irrecoverable full function transactions • Recoverable transactions are rejected by IMS	• Fastpath transactions • Conversational transactions • Full function transactions
Note: IMS commands cannot use persistent WebSphere MQ messages with commit mode 0. See the *IMS/ESA Open Transaction Manager Access User's Guide* for more information.		

If the message cannot be put to the IMS queue:

If the message cannot be put to the IMS queue, the following action is taken by WebSphere MQ:

- If a message cannot be put to IMS because the message is invalid, the message is put to the dead-letter queue and a message is sent to the system console.
- If the message is valid, but is rejected by IMS, WebSphere MQ sends an error message to the system console, the message includes the IMS sense code, and the WebSphere MQ message is put to the dead-letter queue. If the IMS sense code is 001A, IMS sends a WebSphere MQ message containing the reason for the failure to the reply-to queue.

  Note: In the circumstances listed above, if WebSphere MQ cannot put the message to the dead-letter queue for any reason, the message is returned to the originating WebSphere MQ queue. An error message is sent to the system console, and no further messages are sent from that queue.

  To resend the messages, do ***one*** of the following:
  - Stop and restart the Tpipes in IMS corresponding to the queue
  - Alter the queue to GET(DISABLED), and again to GET(ENABLED)
  - Stop and restart IMS or the OTMA
  - Stop and restart your WebSphere MQ subsystem
- If the message is rejected by IMS for anything other than a message error, the WebSphere MQ message is returned to the originating queue, WebSphere MQ stops processing the queue, and an error message is sent to the system console.

  If an exception report message is required, the bridge puts it to the reply-to queue with the authority of the originator. If the message cannot be put to the queue, the report message is put to the dead-letter queue with the authority of the bridge. If it cannot be put to the DLQ, it is discarded.

IMS bridge feedback codes:

IMS sense codes are normally output in hexadecimal format in WebSphere MQ console messages such as CSQ2001I (for example, sense code 001A). WebSphere MQ feedback codes as seen in the dead-letter header of messages put to the dead-letter queue are decimal numbers.

The IMS bridge feedback codes are in the range 301 through 399. They are mapped from the IMS-OTMA sense codes as follows:

1. The IMS-OTMA sense code is converted from a hexadecimal number to a decimal number.
2. 300 is added to the number resulting from the calculation in 1, giving the WebSphere MQ *Feedback* code.

Refer to the *IMS/ESA Open Transaction Manager Access Guide* for information about IMS-OTMA sense codes.

The MQMD fields in messages from the IMS bridge:

The MQMD of the originating message is carried by IMS in the User Data section of the OTMA headers. If the message originates in IMS, this is built by the IMS Destination Resolution Exit. The MQMD of a message received from IMS is built as follows:

StrucID
"MD "

Version
MQMD_VERSION_1

Report
MQRO_NONE

MsgType
MQMT_REPLY

Expiry If MQIIH_PASS_EXPIRATION is set in the Flags field of the MQIIH, this field contains the remaining expiry time, else it is set to MQEI_UNLIMITED

Feedback
MQFB_NONE

Encoding
MQENC.Native (the encoding of the z/OS system)

CodedCharSetId
MQCCSI_Q_MGR (the CodedCharSetID of the z/OS system)

Format
MQFMT_IMS if the MQMD.Format of the input message is MQFMT_IMS, otherwise IOPCB.MODNAME

Priority
MQMD.Priority of the input message

Persistence
Depends on commit mode: MQMD.Persistence of the input message if CM-1; persistence matches recoverability of the IMS message if CM-0

MsgId MQMD.MsgId if MQRO_PASS_MSG_ID, otherwise New MsgId (the default)

CorrelId
MQMD.CorrelId from the input message if MQRO_PASS_CORREL_ID, otherwise MQMD.MsgId from the input message (the default)

BackoutCount
0

ReplyToQ
Blanks

ReplyToQMgr
Blanks (set to local qmgr name by the queue manager during the MQPUT)

UserIdentifier
MQMD.UserIdentifier of the input message

AccountingToken
MQMD.AccountingToken of the input message

ApplIdentityData
MQMD.ApplIdentityData of the input message

PutApplType
MQAT_XCF if no error, otherwise MQAT_BRIDGE

PutApplName
<XCFgroupName><XCFmemberName> if no error, otherwise QMGR name

PutDate
Date when message was put

PutTime
Time when message was put

ApplOriginData
Blanks

The MQIIH fields in messages from the IMS bridge:

The MQIIH of a message received from IMS is built as follows:

StrucId
″IIH ″

Version
1

StrucLength
84

Encoding
MQENC_NATIVE

CodedCharSetId
MQCCSI_Q_MGR

Format
MQIIH.ReplyToFormat of the input message if MQIIH.ReplyToFormat is not blank, otherwise IOPCB.MODNAME

Flags 0

LTermOverride
LTERM name (Tpipe) from OTMA header

MFSMapName
Map name from OTMA header

ReplyToFormat
Blanks

Authenticator
MQIIH.Authenticator of the input message if the reply message is being put to an MQ-IMS bridge queue, otherwise blanks.

TranInstanceId
Conversation ID / Server Token from OTMA header if in conversation, otherwise nulls

TranState
″C″ if in conversation, otherwise blank

CommitMode
Commit mode from OTMA header (″0″ or ″1″)

SecurityScope
Blank

Reserved
Blank

Reply messages from IMS:

When an IMS transaction ISRTs to its IOPCB, the message is routed back to the originating LTERM or TPIPE.

These are seen in WebSphere MQ as reply messages. Reply messages from IMS are put onto the reply-to queue specified in the original message. If the message cannot be put onto the reply-to queue, it is put onto the dead-letter queue using the authority of the bridge. If the message cannot be put onto the dead-letter queue, a negative acknowledgement is sent to IMS to say that the message cannot be received. Responsibility for the message is then returned to IMS. If you are using commit mode 0, messages from that Tpipe are not sent to the bridge, and remain on the IMS queue; that is, no further messages are sent until restart. If you are using commit mode 1, other work can continue.

If the reply has an MQIIH structure, its format type is MQFMT_IMS; if not, its format type is specified by the IMS MOD name used when inserting the message.

Using alternate response PCBs in IMS transactions:

When an IMS transaction uses alternate response PCBs (ISRTs to the ALTPCB, or issues a CHNG call to a modifiable PCB), the pre-routing exit (DFSYPRX0) is invoked to determine if the message should be rerouted.

If the message is to be rerouted, the destination resolution exit (DFSYDRU0) is invoked to confirm the destination and prepare the header information See the *WebSphere MQ for z/OS System Setup Guide* for information about these exit programs.

Unless action is taken in the exits, all output from IMS transactions initiated from a WebSphere MQ queue manager, whether to the IOPCB or to an ALTPCB, will be returned to the same queue manager.

Sending unsolicited messages from IMS:

To send messages from IMS to a WebSphere MQ queue, you need to invoke an IMS transaction that ISRTs to an ALTPCB.

You need to write pre-routing and destination resolution exits to route unsolicited messages from IMS and build the OTMA user data, so that the MQMD of the message can be built correctly. See the *WebSphere MQ for z/OS System Setup Guide* for information about these exit programs.

Note: The WebSphere MQ-IMS bridge does not know whether a message that it receives is a reply or an unsolicited message. It handles the message the same way in each case, building the MQMD and MQIIH of the reply based on the OTMA UserData that arrived with the message

Unsolicited messages can create new Tpipes. For example, if an existing IMS transaction switched to a new LTERM (for example PRINT01), but the implementation requires that the output be delivered through OTMA, a new Tpipe (called PRINT01 in this example) is created. By default, this is a non-synchronized Tpipe. If the implementation requires the message to be recoverable, set the destination resolution exit output flag. See the *IMS Customization Guide* for more information.

Message segmentation:

You can define IMS transactions as expecting single- or multi-segment input.

The originating WebSphere MQ application must construct the user input following the MQIIH structure as one or more LLZZ-data segments. All segments of an IMS message must be contained in a single WebSphere MQ message sent with a single MQPUT.

The maximum length of an LLZZ-data segment is defined by IMS/OTMA (32764 bytes). The total WebSphere MQ message length is the sum of the LL bytes, plus the length of the MQIIH structure.

All the segments of the reply are contained in a single WebSphere MQ message.

There is a further restriction on the 32 KB limitation on messages with format MQFMT_IMS_VAR_STRING. When the data in an ASCII-mixed CCSID message is converted to an EBCDIC-mixed CCSID message, a shift-in byte or a shift-out byte is added every time that there is a transition between SBCS and DBCS characters. The 32 KB restriction applies to the maximum size of the message. That is, because the LL field in the message cannot exceed 32 KB, the message must not exceed 32 KB including all shift-in and shift-out characters. The application building the message must allow for this.

Data conversion:

The data conversion (including calling any necessary exits) is performed by the distributed queuing facility when it puts a message to a destination queue that has XCF information defined for its storage class.

Any exits needed must be available to the distributed queuing facility in the data set referenced by the CSQXLIB DD statement. This means that you can send messages to an IMS application using the WebSphere MQ-IMS bridge from any WebSphere MQ platform.

Note: Because the WebSphere MQ-IMS bridge does not convert messages when it gets a message, messages arriving through the CICS distributed queuing facility are not converted.

If there are conversion errors, the message is put to the queue unconverted; this results eventually in it being treated as an error by the WebSphere MQ-IMS bridge, because the bridge cannot recognize the header format. If a conversion error occurs, an error message is sent to the z/OS console.

See “Writing data-conversion exits” on page 163 for detailed information about data conversion in general.

Sending messages to the WebSphere MQ-IMS bridge:

To ensure that conversion is performed correctly, you must tell the queue manager what the format of the message is.

If the message has an MQIIH structure, the *Format* in the MQMD must be set to the built-in format MQFMT_IMS, and the *Format* in the MQIIH must be set to the name of the format that describes your message data. If there is no MQIIH, set the *Format* in the MQMD to your format name.

If your data (other than the LLZZs) is all character data (MQCHAR), use as your format name (in the MQIIH or MQMD, as appropriate) the built-in format MQFMT_IMS_VAR_STRING. Otherwise, use your own format name, in which case

you must also provide a data-conversion exit for your format. The exit must handle the conversion of the LLZZs in your message, in addition to the data itself (but it does not have to handle any MQIIH at the start of the message).

If your application uses *MFSMapName*, you are recommended to use messages with the MQFMT_IMS instead, and define the map name passed to the IMS transaction in the MFSMapName field of the MQIIH.

Receiving messages from the WebSphere MQ-IMS bridge:

If an MQIIH structure is present on the original message that you are sending to IMS, one is also present on the reply message.

To ensure that your reply is converted correctly:

- If you have an MQIIH structure on your original message, specify the format that you want for your reply message in the MQIIH *ReplytoFormat* field of the original message. This value is placed in the MQIIH *Format* field of the reply message. This is particularly useful if all your output data is of the form LLZZ<character data>.
- If you do not have an MQIIH structure on your original message, specify the format that you want for the reply message as the MFS MOD name in the IMS application's ISRT to the IOPCB.

Writing your program

The coding required to handle IMS transactions through WebSphere MQ depends on the message format required by the IMS transaction and the range of responses it can return. However, there are several points to consider when your application handles IMS screen formatting information.

When an IMS transaction is started from a 3270 screen, the message passes through IMS Message Format Services. This can remove all terminal dependency from the data stream seen by the transaction. When a transaction is started through OTMA, MFS is not involved. If application logic is implemented in MFS, this must be re-created in the new application.

In some IMS transactions, the end-user application can modify certain 3270 screen behavior, for example, highlighting a field that has had invalid data entered. This type of information is communicated by adding a two-byte attribute field to the IMS message for each screen field that needs to be modified by the program.

Thus, if you are coding an application to mimic a 3270, you need to take account of these fields when building or receiving messages.

You might need to code information in your program to process:

- Which key is pressed (for example, Enter and PF1)
- Where the cursor is when the message is passed to your application
- Whether the attribute fields have been set by the IMS application
  - High, normal, or zero intensity
  - Color
  - Whether IMS is expecting the field back the next time that Enter is pressed
- Whether the IMS application has used null characters (X'3F') in any fields.

If your IMS message contains only character data (apart from the LLZZ-data segment), and you are using an MQIIH structure, set the MQMD format to MQFMT_IMS and the MQIIH format to MQFMT_IMS_VAR_STRING.

If your IMS message contains only character data (apart from the LLZZ-data segment), and you are ***not*** using an MQIIH structure, set the MQMD format to MQFMT_IMS_VAR_STRING and ensure that your IMS application specifies MODname MQFMT_IMS_VAR_STRING when replying. If a problem occurs (for example, user not authorized to use the transaction) and IMS sends an error message, this has an MODname of the form DFSMOx, where x is a number between 1 and 5. This is put in the MQMD.Format.

If your IMS message contains binary, packed, or floating point data (apart from the LLZZ-data segment), code your own data-conversion routines. Refer to *IMS/ESA Application Programming: Transaction Manager* for information about IMS screen formatting.

Writing WebSphere MQ applications to invoke IMS conversational transactions:

When you write an application that invokes an IMS conversation, consider the following:

- Include an MQIIH structure with your application message.
- Set the *CommitMode* in MQIIH to MQICM_SEND_THEN_COMMIT.
- To invoke a new conversation, set *TranState* in MQIIH to MQITS_NOT_IN_CONVERSATION.
- To invoke second and subsequent steps of a conversation, set *TranState* to MQITS_IN_CONVERSATION, and set *TranInstanceId* to the value of that field returned in the previous step of the conversation.
- There is no easy way in IMS to find the value of a *TranInstanceId*, should you lose the original message sent from IMS.
- The application must check the *TranState* of messages from IMS to check whether the IMS transaction has terminated the conversation.
- You can use /EXIT to end a conversation. You must also quote the *TranInstanceId*, set *TranState* to MQITS_IN_CONVERSATION, and use the WebSphere MQ queue on which the conversation is being carried out.
- You cannot use /HOLD or /REL to hold or release a conversation.
- Conversations invoked through the WebSphere MQ-IMS bridge are terminated if IMS is restarted.

Triggering:

The WebSphere MQ-IMS bridge does not support trigger messages.

If you define an initiation queue that uses a storage class with XCF parameters, messages put to that queue are rejected when they get to the bridge.

Writing programs containing IMS commands:

An application program can build a WebSphere MQ message of the form LLZZ<command>, instead of a transaction, where <command> is of the form /DIS TRAN PART or /DIS POOL ALL.

Most IMS commands can be issued in this way; see the *IMS/ESA V6 OTMA Guide and Reference* for details. The command output is received in the WebSphere MQ reply message in the text form as would be sent to a 3270 terminal for display.

OTMA has implemented a special form of the IMS display transaction command, which returns an architected form of the output. The exact format is defined in the *IMS/ESA V6 OTMA Guide and Reference*. To invoke this form from a WebSphere MQ message, build the message data as before, for example /DIS TRAN PART, and set the TranState field in the MQIIH to MQITS_ARCHITECTED. IMS processes the command, and returns the reply in the architected form. An architected response contains all the information that could be found in the text form of the output, and one additional piece of information: whether the transaction is defined as recoverable or non-recoverable.

Object-oriented programming with WebSphere MQ

WebSphere MQ provides two ways of programming WebSphere MQ applications that can be used with object-oriented programming languages.

The *WebSphere MQ Object Model* providesclasses that provide the same functionality as WebSphere MQ calls and structures, but that are a more natural way of programming in an object-oriented environment.

Websphere MQ also provides classes that implement the Java Message Service (JMS) specification. For details of the Websphere MQ classes for JMS, see *WebSphere MQ Using Java*. Message Service Clients for C/C++ and .NET provide an application programming Interface (API) called XMS that has the same set of interfaces as the Java Message Service (JMS) API.

What is in the WebSphere MQ Object Model?

The WebSphere MQ Object Model consists of the following:

- *Classes* representing familiar WebSphere MQ concepts such as queue managers, queues, and messages.
- *Methods* on each class corresponding to MQI calls.
- *Properties* on each class corresponding to attributes of WebSphere MQ objects.

When creating a WebSphere MQ application using the WebSphere MQ Object Model, you create instances of these classes in the program. An instance of a class in object-oriented programming is called an *object*. When an object has been created, you interact with the object by examining or setting the values of the object's properties (the equivalent of issuing an MQINQ or MQSET call), and by making method calls against the object (the equivalent of issuing the other MQI calls).

Classes

The WebSphere MQ Object Model provides the following base set of classes.

The actual implementation of the model varies slightly between the different supported object-oriented environments.

MQQueueManager

An object of the MQQueueManager class represents a connection to a queue manager. It has methods to Connect(), Disconnect(), Commit(), and Backout() (the equivalent of MQCONN or MQCONNX, MQDISC,

MQCMIT, and MQBACK). It has properties corresponding to the attributes of a queue manager. Accessing a queue manager attribute property implicitly connects to the queue manager if not already connected. Destroying an MQQueueManager object implicitly disconnects from the queue manager.

MQQueue
An object of the MQQueue class represents a queue. It has methods to Put() and Get() messages to and from the queue (the equivalent of MQPUT and MQGET). It has properties corresponding to the attributes of a queue. Accessing a queue attribute property, or issuing a Put() or Get() method call, implicitly opens the queue (the equivalent of MQOPEN). Destroying an MQQueue object implicitly closes the queue (the equivalent of MQCLOSE).

MQTopic
An object of the MQTopic class represents a topic. It has methods to Put() (publish) and Get() (receive or subscribe) messages to and from the topic (the equivalent of MQPUT and MQGET). It has properties corresponding to the attributes of a topic. An MQTopic object can only be accessed for publication or subscription, not both simultaneously. When used for receiving messages the MQTopic object can be created with an unmanaged or managed subscription and as a durable or non-durable subscriber - multiple overloaded constructors are provided for these differing scenarios.

MQMessage
An object of the MQMessage class represents a message to be put on a queue or got from a queue. It contains a buffer, and encapsulates both application data and MQMD. It has properties corresponding to MQMD fields, and methods that allow you to write and read user data of different types (for example, strings, long integers, short integers, single bytes) to and from the buffer.

MQPutMessageOptions
An object of the MQPutMessageOptions class represents the MQPMO structure. It has properties corresponding to MQPMO fields.

MQGetMessageOptions
An object of the MQGetMessageOptions class represents the MQGMO structure. It has properties corresponding to MQGMO fields.

MQProcess
An object of the MQProcess class represents a process definition (used with triggering). It has properties that represent the attributes of a process definition.

MQDistributionList
(**Not WebSphere MQ for z/OS.**) An object of the MQDistributionList class represents a distribution list (used to send multiple messages with a single MQPUT). It contains a list of MQDistributionListItem objects.

MQDistributionListItem
(**Not WebSphere MQ for z/OS.**) An object of the MQDistributionListItem class represents a single distribution list destination. It encapsulates the MQOR, MQRR, and MQPMR structures, and has properties corresponding to the fields of these structures.

Object references

In a WebSphere MQ program that uses the MQI, WebSphere MQ returns connection handles and object handles to the program.

These handles must be passed as parameters on subsequent WebSphere MQ calls. With the WebSphere MQ Object Model, these handles are hidden from the application program. Instead, the creation of an object from a class results in an object reference being returned to the application program. It is this object reference that is used when making method calls and property accesses against the object.

Return codes

Issuing a method call or setting a property value results in return codes being set.

These return codes are a completion code and a reason code, and are themselves properties of the object. The values of completion code and reason code are the same as those defined for the MQI, with some extra values specific to the object-oriented environment.

Programming language considerations

The WebSphere MQ Object Model is implemented in C++, Java, and for the .NET environment.

See *WebSphere MQ Using C++* for information about coding programs using the WebSphere MQ Object Model in C++.

See *WebSphere MQ Using Java* for information about coding programs using the WebSphere MQ Object Model in Java.

See *WebSphere MQ Using .NET* for information about coding .NET programs using the WebSphere MQ .NET classes.

Coding in ActiveX

Support for ActiveX has been stabilized at the WebSphere MQ Version 6.0 level. To exploit features introduced to WebSphere MQ later than Version 6.0, consider using .NET instead.

Refer to *WebSphere MQ Using the Component Object Model Interface* for information about coding programs using the WebSphere MQ Object Model in ActiveX.

The WebSphere MQ ActiveX is commonly known as the MQAX. The MQAX is included as part of WebSphere MQ for Windows.

Chapter 3. Building a WebSphere MQ application

Building your application on AIX

The AIX publications describe how to build executable applications from the programs that you write.

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for AIX applications to run under AIX. C, C++, and COBOL are supported. For information about preparing your C++ programs, see WebSphere MQ Using C++.

The tasks that you must perform to create an executable application using WebSphere MQ for AIX vary with the programming language that your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the WebSphere MQ for AIX include files for the language that you are using. Make yourself familiar with the contents of these files. See Chapter 9, "WebSphere MQ data definition files," on page 575 for a full description.

When you run threaded server or threaded client applications, set the environment variable AIXTHREAD_SCOPE=S.

Preparing C programs

Precompiled C programs are supplied in the /usr/mqm/samp/bin directory. Use the ANSI compiler and run the following commands. For further information on programming 64 bit applications see Chapter 10, "Coding standards on 64 bit platforms," on page 579.

For 32–bit applications:

```
$ cc -o amqsput_32 amqsput0.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqm
```

where `amqsput0` is a sample program.

For 64–bit applications:

```
$ cc -q64 -o amqsput_64 amqsput0.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqm
```

where `amqsput0` is a sample program.

If you are using the VisualAge® C/C++ v6.0 compiler for C++ programs you must include the option `-q namemangling=v5` to get all the WebSphere MQ symbols resolved when linking the libraries.

If you want to use the programs on a machine that has only the WebSphere MQ client for AIX installed, recompile the programs to link them with the client library (`-lmqic`) instead.

Linking libraries

You need the following libraries:

- Link your programs with the appropriate library provided by WebSphere MQ.

In a non-threaded environment, link to one of the following libraries:

Library file	Program/exit type
libmqm.a	Server for C
libmqic.a	Client for C

In a threaded environment, link to one of the following libraries:

Library file	Program/exit type
libmqm_r.a	Server for C
libmqic_r.a	Client for C

For example, to build a simple threaded WebSphere MQ application from a single compilation unit run the following commands.

For 32–bit applications:

```
$ xlc_r -o amqsputc_32_r amqsput0.c -I/usr/mqm/inc -L/usr/mqm/lib -lmqm_r
```

where `amqsput0` is a sample program.

For 64–bit applications:

```
$ xlc_r -q64 -o amqsputc_64_r amqsput0.c -I/usr/mqm/inc -L/usr/mqm/lib64 -lmqm_r
```

where `amqsput0` is a sample program.

If you want to use the programs on a machine that has only the WebSphere MQ client for AIX installed, recompile the programs to link them with the client library (`-lmqic`) instead.

Note:

1. If you are writing an installable service (see the WebSphere MQ System Administration Guide for further information), you need to link to the libmqmzf.a library in a non-threaded application and to the libmqmzf_r.a library in a threaded application.
2. If you are producing an application for external coordination by an XA-compliant transaction manager such as IBM TXSeries, Encina, or BEA Tuxedo, you need to link to the libmqmxa.a (or libmqmxa64.a if your transaction manager treats the 'long' type as 64-bit) and libmqz.a libraries in a non-threaded application and to the libmqmxa_r.a (or libmqmxa64_r.a) and libmqz_r.a libraries in a threaded application.
3. You need to link trusted applications to the threaded WebSphere MQ libraries. However, only one thread in a trusted application on WebSphere MQ on UNIX systems can be connected at a time.
4. You must link WebSphere MQ libraries before any other product libraries.

Preparing COBOL programs

Notes® to users

1. 32 bit COBOL copy books are installed in the following directory:

   `/usr/mqm/inc/cobcpy32`

   and symbolic links are created in:

   `/usr/mqm/inc`

2.

64 bit COBOL copy books are installed in the following directory:
`/usr/mqm/inc/cobcpy64`

3. In the following examples set `COBCPY` to:
   `/usr/mqm/inc/cobcpy32`

   for 32 bit applications, and:
   `/usr/mqm/inc/cobcpy64`

   for 64 bit applications.

You need to link your program with one of the following:

Library file	Program/exit type
libmqmcb.a	Server for COBOL
libmqicb.a	Client for COBOL
libmqmcb_r	Server for COBOL (threaded application)

You can use the IBM COBOL Set compiler or Micro Focus COBOL compiler depending on the program:

- Programs beginning `amqm` are suitable for the Micro Focus COBOL compiler, and
- Programs beginning `amq0` are suitable for either compiler.

Preparing COBOL programs using IBM COBOL Set for AIX

Sample COBOL programs are supplied with WebSphere MQ. To compile such a program, enter the appropriate command from the list below:

32 bit non-threaded server application

```
$ cob2 -o amq0put0 amq0put0.cbl -L/usr/mqm/lib -lmqmcb -qLIB \
-I<COBCPY>
```

32 bit non-threaded client application

```
$ cob2 -o amq0put0 amq0put0.cbl -L /usr/mqm/lib -lmqicb -qLIB \
-I<COBCPY>
```

32 bit threaded server application

```
$ cob2_r -o amq0put0 amq0put0.cbl -qTHREAD -L/usr/mqm/lib \
-lmqmcb_r -qLIB -I<COBCPY>
```

32 bit threaded client application

```
$ cob2_r -o amq0put0 amq0put0.cbl -qTHREAD -L /usr/mqm/lib \
-lmqicb_r -qLIB -I<COBCPY>
```

Preparing COBOL programs using Micro Focus COBOL

Set environment variables before compiling your program as follows:

```
export COBCPY=<COBCPY>
export LIB=/usr/mqm/lib:$LIB
```

To compile a 32 bit COBOL program using Micro Focus COBOL, enter:

```
$ cob32 -xvP amqsput.cbl -L /usr/mqm/lib -lmqmcb Server for COBOL
$ cob32 -xvP amqsput.cbl -L /usr/mqm/lib -lmqicb Client for COBOL
$ cob32 -xtvP amqsput.cbl -L /usr/mqm/lib -lmqmcb_r Threaded Server for COBOL
$ cob32 -xtvP amqsput.cbl -L /usr/mqm/lib -lmqicb_r Threaded Client for COBOL
```

To compile a 64 bit COBOL program using Micro Focus COBOL, enter:

```
$ cob64 -xvP amqsput.cbl -L /usr/mqm/lib64 -lmqmcb Server for COBOL
$ cob64 -xvP amqsput.cbl -L /usr/mqm/lib64 -lmqicb Client for COBOL
$ cob64 -xtvP amqsput.cbl -L /usr/mqm/lib64 -lmqmcb_r Threaded Server for COBOL
$ cob64 -xtvP amqsput.cbl -L /usr/mqm/lib64 -lmqicb_r Threaded Client for COBOL
```

where amqsput is a sample program

See the Micro Focus COBOL documentation for a description of the environment variables that you need to set up.

Preparing CICS programs

XA switch modules are provided to enable you to link CICS with WebSphere MQ:

Table 17. Essential code for CICS applications (AIX)

Description	C (source)	C (exec) - add to your XAD.Stanza
XA initialization routine	amqzscix.c	amqzsc - CICS for AIX

You are recommended to use the prebuilt version of amqzsc that is shipped with the product. If you need to rebuild the switch load file for any reason (for example, a new CICS release might require this), then you can do it as follows:

```
export MQM_HOME=/usr/mqm

echo "amqzscix" > tmp.exp
xlc_r4 $MQM_HOME/samp/amqzscix.c -I/usr/lpp/encina/include \
  -e amqzscix -bE:tmp.exp -bM:SRE -o amqzsc                \
  /usr/lpp/cics/lib/regxa_swxa.o -L$MQM_HOME/lib           \
  -L/usr/lpp/cics/lib -L/usr/lpp/encina/lib                \
  -lcicsrt -lEncina -lEncServer -lpthreads -lc_r           \
  -lmqmcics_r -lmqmxa_r -lmqz_r -lmqmcs_r -lmqmzse
rm tmp.exp
```

Always link your C transactions with the threadsafe WebSphere MQ library libmqm_r.a., and your COBOL transactions with the COBOL library libmqmcb_r.a.

You can find more information about supporting CICS transactions in the WebSphere MQ System Administration Guide.

TXSeries CICS support

WebSphere MQ on AIX supports TXSeries CICS using the XA interface. Ensure that CICS applications are linked to the threaded version of the MQ libraries.

You can run CICS programs using IBM COBOL Set for AIX or Micro Focus COBOL. The following sections describe the difference between these.

Preparing CICS COBOL programs using IBM COBOL Set for AIX:

To use IBM COBOL, follow these steps:

1. Export the following environment variable:

   ```
   export LDFLAGS="-qLIB -bI:/usr/lpp/cics/lib/cicsprIBMCOB.exp  \
                   -I/usr/mqm/inc -I/usr/lpp/cics/include  \
                   -e _iwz_cobol_main  \
   ```

   where LIB is a compiler directive.

2. Translate, compile, and link the program by typing:
   ```
   cicstcl -l IBMCOB <yourprog>.ccp
   ```

Preparing CICS COBOL programs using Micro Focus COBOL:

To use Micro Focus COBOL, follow these steps:

1. Add the WebSphere MQ COBOL run-time library module to the run-time library using the following command:
   ```
   cicsmkcobol -L/usr/lib/dce -L/usr/mqm/lib  \
               /usr/mqm/lib/libmqmcbrt.o -lmqz_r
   ```

   Note: With `cicsmkcobol`, WebSphere MQ does not allow you to make MQI calls in the C programming language from your COBOL application.

   If your existing applications have any such calls, you are strongly recommended to move these functions from the COBOL applications to your own library, for example, `myMQ.so`. After you have done this, do not include the WebSphere MQ library `libmqmcbrt.o` when building the COBOL application for CICS.

   Additionally, if your COBOL application does not make any COBOL MQI call , do not link `libmqmz_r` with `cicsmkcobol`.

   This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call WebSphere MQ on UNIX systems.

   Note: Run `cicsmkcobol` only when you install one of the following:
   - New version or release of Micro Focus COBOL
   - New version or release of CICS for AIX
   - New version or release of any supported database product (for COBOL transactions only)
   - New version or release of WebSphere MQ
2. Export the following environment variable:
   ```
   COBCPY=/usr/mqm/inc  export COBCPY
   ```
3. Translate, compile, and link the program by typing:
   ```
   cicstcl -l COBOL -e <yourprog>.ccp
   ```

Preparing CICS C programs:

Build CICS C programs using the standard CICS facilities:

1. Export ***one*** of the following environment variables:
   - LDFLAGS = "-L/usr/mqm/lib -lmqm_r" export LDFLAGS
   - USERLIB = "-L/usr/mqm/lib -lmqm_r" export USERLIB
2. Translate, compile, and link the program by typing:
   ```
   cicstcl -l C amqscic0.ccs
   ```

CICS C sample transaction:

Sample C source for a CICS WebSphere MQ transaction is provided by AMQSCIC0.CCS. The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue whose name is contained in the transmission header of

the message. Any failures are sent to the queue SYSTEM.SAMPLE.CICS.DLQ. Use the sample MQSC script AMQSCIC0.TST to create these queues and sample input queues.

Building your application on HP-UX

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for HP-UX applications to run under HP-UX.

C, C++, and COBOL are supported. For information about preparing your C++ programs, see WebSphere MQ Using C++.

The tasks that you must perform to create an executable application using WebSphere MQ for HP-UX vary with the programming language that your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the WebSphere MQ for HP-UX include files for the language that you are using. Make yourself familiar with the contents of these files. See Chapter 9, “WebSphere MQ data definition files,” on page 575 for a full description.

Throughout this chapter, we use the \ character to split long commands over more than one line. Do not enter this character; enter each command as a single line.

Preparing C programs

Work in your normal environment. Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. For further information on programming 64 bit applications see Chapter 10, “Coding standards on 64 bit platforms,” on page 579.

To use SSL, WebSphere MQ clients on HP-UX 11i v1 and HP-UX 11i v2 must be built:

- Using the C++ compiler (not the C compiler)
- Using POSIX threads
- With the compiler options: -W1,+b/opt/ibm/gsk7/lib:/opt/mqm/lib

PA-RISC platform

Build examples of amqsput0, cliexit and srvexit on PA-RISC platform.

The following is an example of how to build the sample program amqsput0 as a client application in a non-threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib -lmqic
```

The following is an example of how to build the sample program amqsput0 as a client application in a threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib -lmqic_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a client application in a non-threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic
```

The following is an example of how to build the sample program amqsput0 as a client application in a threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a server application in a non-threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib -lmqm
```

The following is an example of how to build the sample program amqsput0 as a server application in a threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib -lmqm_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a server application in a non-threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqm
```

The following is an example of how to build the sample program amqsput0 as a server application in a threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqm_r -lpthread
```

The following is an example of how to build a client exit cliexit in a non-threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld +b: -b cliexit.o +ee MQStart -o /var/mqm/exits/cliexit_32  -L/opt/mqm/lib \
  -L/usr/lib -lmqic
```

The following is an example of how to build a client exit cliexit in a threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld +b: -b cliexit.o +ee MQStart -o /var/mqm/exits/cliexit_32_r  -L/opt/mqm/lib \
  -L/usr/lib -lmqic_r -lpthread
```

The following is an example of how to build a client exit cliexit in a non-threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld -b +noenvvar cliexit.o +ee MQStart -o /var/mqm/exits64/cliexit_64  \
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic
```

The following is an example of how to build a client exit cliexit in a threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld -b +noenvvar cliexit.o +ee MQStart -o /var/mqm/exits64/cliexit_64_r  \
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic_r -lpthread
```

The following is an example of how to build a server exit srvexit in a non-threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld +b: -b srvexit.o +ee MQStart -o /var/mqm/exits/srvexit_32  -L/opt/mqm/lib \
  -L/usr/lib -lmqic
```

The following is an example of how to build a server exit srvexit in a threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld +b: -b srvexit.o +ee MQStart -o /var/mqm/exits/srvexit_32_r  -L/opt/mqm/lib \
  -L/usr/lib -lmqic_r -lpthread
```

The following is an example of how to build a server exit srvexit in a non-threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld -b +noenvvar srvexit.o +ee MQStart -o /var/mqm/exits64/srvexit_64  \
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic
```

The following is an example of how to build a server exit srvexit in a threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld -b +noenvvar srvexit.o +ee MQStart -o /var/mqm/exits64/srvexit_64_r  \
  -L/opt/mqm/lib64 -L/usr/lib/pa20_64 -lmqic_r -lpthread
```

IA64 (IPF) platform

The following is an example of how to build the sample program amqsput0 as a client application in a non-threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsputc_32 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib/hpux32 -lmqic
```

The following is an example of how to build the sample program amqsput0 as a client application in a threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsputc_32_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib/hpux32 -lmqic_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a client application in a non-threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsputc_64 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqic
```

The following is an example of how to build the sample program amqsput0 as a client application in a threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsputc_64_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqic_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a server application in a non-threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib/hpux32 -lmqm
```

The following is an example of how to build the sample program amqsput0 as a server application in a threaded 32-bit environment:

```
c89 -Wl,+b,: +e -D_HPUX_SOURCE -o amqsput_32_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib -L/usr/lib/hpux32 -lmqm_r -lpthread
```

The following is an example of how to build the sample program amqsput0 as a server application in a non-threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64 amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqm
```

The following is an example of how to build the sample program amqsput0 as a server application in a threaded 64-bit environment:

```
c89 +DD64 +e -Wl,+noenvvar -D_HPUX_SOURCE -o amqsput_64_r amqsput0.c -I/opt/mqm/inc
  -L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqm_r -lpthread
```

The following is an example of how to build a client exit cliexit in a non-threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld +b: -b cliexit.o +ee MQStart -o /var/mqm/exits/cliexit_32 -L/opt/mqm/lib \
-L/usr/lib/hpux32 -lmqic
```

The following is an example of how to build a client exit cliexit in a threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld +b: -b cliexit.o +ee MQStart -o /var/mqm/exits/cliexit_32_r -L/opt/mqm/lib \
-L/usr/lib/hpux32 -lmqic_r -lpthread
```

The following is an example of how to build a client exit cliexit in a non-threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld -b +noenvvar cliexit.o +ee MQStart -o /var/mqm/exits64/cliexit_64 \
-L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqic
```

The following is an example of how to build a client exit cliexit in a threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o cliexit.o cliexit.c -I/opt/mqm/inc
ld  -b +noenvvar cliexit.o +ee MQStart -o /var/mqm/exits/cliexit_64_r \
-L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqic_r -lpthread
```

The following is an example of how to build a server exit srvexit in a non-threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld +b: -b srvexit.o +ee MQStart -o /var/mqm/exits/srvexit_32 -L/opt/mqm/lib \
-L/usr/lib/hpux32 -lmqm
```

The following is an example of how to build a server exit srvexit in a threaded 32-bit environment:

```
c89 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld +b: -b srvexit.o +ee MQStart -o /var/mqm/exits/srvexit_32_r -L/opt/mqm/lib \
-L/usr/lib/hpux32 -lmqm_r -lpthread
```

The following is an example of how to build a server exit srvexit in a non-threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld -b +noenvvar srvexit.o +ee MQStart -o /var/mqm/exits64/srvexit_64 \
-L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqm
```

The following is an example of how to build a server exit srvexit in a threaded 64-bit environment:

```
c89 +DD64 +e +z -c -D_HPUX_SOURCE -o srvexit.o srvexit.c -I/opt/mqm/inc
ld  -b +noenvvar srvexit.o +ee MQStart -o /var/mqm/exits/srvexit_64_r \
-L/opt/mqm/lib64 -L/usr/lib/hpux64 -lmqm_r -lpthread
```

Linking libraries

You need to link your programs with the appropriate library provided by WebSphere MQ.

The following table shows which library to use in different environments

Hardware platform	Threaded or non-threaded environment	Program/exit type	Library file
PA-RISC	Threaded	Server for C	libmqm.sl

Hardware platform	Threaded or non-threaded environment	Program/exit type	Library file
PA-RISC	Threaded	Client for C	libmqic.sl
PA-RISC	Non-threaded	Server for C	libmqm_r.sl
PA-RISC	Non-threaded	Client for C	libmqic_r.sl
IA64 (IPF)	Threaded	Server for C	libmqm.so
IA64 (IPF)	Threaded	Client for C	libmqic.so
IA64 (IPF)	Non-threaded	Server for C	libmqm_r.so
IA64 (IPF)	Non-threaded	Client for C	libmqic_r.so

Note:

1. If you are writing an installable service (see the WebSphere MQ System Administration Guide for further information), you need to link to the libmqmzf.sl library.
2. If you are producing an application for external coordination by an XA-compliant transaction manager such as IBM TXSeries Encina, or BEA Tuxedo, you need to link to the libmqmxa.sl (or libmqmxa64.sl if your transaction manager treats the 'long' type as 64-bit) and libmqz.sl libraries in a non-threaded application and to the libmqmxa_r.sl (or libmqmxa64_r.sl) and libmqz_r.sl libraries in a threaded application.
3. You must link WebSphere MQ libraries before any other product libraries.

Preparing COBOL programs

Notes to users

1. 32 bit COBOL copy books are installed in the following directory:
   `/opt/mqm/inc/cobcpy32`

   and symbolic links are created in:
   `/opt/mqm/inc`
2. 64 bit COBOL copy books are installed in the following directory:
   `/opt/mqm/inc/cobcpy64`
3. In the following examples set `COBCPY` to:
   `/opt/mqm/inc/cobcpy32`

   for 32 bit applications, and:
   `/opt/mqm/inc/cobcpy64`

   for 64 bit applications.

Compile the programs using the Micro Focus compiler. The copy files that declare the structures are in /opt/mqm/inc:

```
$ export LIB=/usr/mqm/lib:$LIB
$ export COBCPY="<COBCPY>"
```

Compiling 32 bit programs:

```
$ cob32 -xv amqsput.cbl -L /opt/mqm/lib -lmqmcb Server for COBOL
$ cob32 -xv amqsput.cbl -L /opt/mqm/lib -lmqicb Client for COBOL
$ cob32 -xtv amqsput.cbl -L /opt/mqm/lib -lmqmcb_r Threaded Server for COBOL
$ cob32 -xtv amqsput.cbl -L /opt/mqm/lib -lmqicb_r Threaded Client for COBOL
```

Compiling 64 bit programs:

```
$ cob64 -xv amqsput.cbl -L /opt/mqm/lib64 -lmqmcb Server for COBOL
$ cob64 -xv amqsput.cbl -L /opt/mqm/lib64 -lmqicb Client for COBOL
$ cob64 -xtv amqsput.cbl -L /opt/mqm/lib64 -lmqmcb_r Threaded Server for COBOL
$ cob64 -xtv amqsput.cbl -L /opt/mqm/lib64 -lmqicb_r Threaded Client for COBOL
```

where `amqsput` is a sample program

Ensure that you have specified adequate run-time stack sizes; 16 KB is the recommended minimum.

You need to link your programs with the appropriate library provided by WebSphere MQ. The following table shows which library to use in different environments

Hardware platform	Program/exit type	Library file
PA-RISC	Server for COBOL	libmqmcb.sl
PA-RISC	Client for COBOL	libmqicb.sl
PA-RISC	Threaded applications	amqmcb_r.sl
IA64 (IPF)	Server for COBOL	libmqmcb.so
IA64 (IPF)	Client for COBOL	libmqicb.so
IA64 (IPF)	Threaded applications	amqmcb_r.so

Using Micro Focus Server Express with WebSphere MQ on the IA64 (IPF) platform

See “Address Space models supported by WebSphere MQ for HP-UX on IA64 (IPF)” on page 351 for details on using Micro Focus Server Express in conjunction with WebSphere MQ on the HP/IPF platform.

Programs to run in the WebSphere MQ client environment

If you are using LU 6.2 to connect your MQI client to a server, link your application to libsna.a, part of the SNAplusAPI product. Use the `–lV3` and `–lstr` options on your compile and link command.

- The `–lV3` option gives your program access to the AT&T signaling library (the SNAplusAPI uses AT&T signals)
- The `–lstr` option links your program to the streams component

If you are not using LU 6.2, consider linking to libsnastubs.a (in /opt/mqm/lib) to fully resolve function names. The need to link to this library varies with how you are using the -B flag during the linking stage.

Preparing CICS programs

To build the sample CICS transaction, amqscic0.ccs, run the following command:

```
$ export USERLIB="-lmqm_r"
$ cicstcl -l C amqscic0.ccs
```

An XA switch module is provided to enable you to link CICS with WebSphere MQ:

Table 18. Essential code for CICS applications (HP-UX)

Description	C (source)	C (exec)
XA initialization routine	amqzscix.c	amqzsc

You can find more information about supporting CICS transactions in the WebSphere MQ System Administration Guide.

TXSeries CICS support

WebSphere MQ on HP-UX supports TXSeries CICS using the XA interface. Ensure that CICS applications are linked to the threaded version of the MQ libraries.

CICS C sample transaction:

Sample C source for a CICS WebSphere MQ transaction is provided by AMQSCIC0.CCS. The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures are sent to the queue SYSTEM.SAMPLE.CICS.DLQ. Use the sample MQSC script AMQSCIC0.TST to create these queues and sample input queues.

Preparing CICS COBOL programs using Micro Focus COBOL:

To use Micro Focus COBOL, follow these steps:

1. Add the WebSphere MQ COBOL run-time library module to the run-time library using the following command:

   ```
   cicsmkcobol -L/usr/lib/dce -L/opt/mqm/lib  \
               /opt/mqm/lib/libmqmcbrt.o -lmqz_r
   ```

   Note: With `cicsmkcobol`, WebSphere MQ does not allow you to make MQI calls in the C programming language from your COBOL application.

   If your existing applications have any such calls, you are strongly recommended to move these functions from the COBOL applications to your own library, for example, `myMQ.so`. After you have done this, do not include the WebSphere MQ library `libmqmcbrt.o` when building the COBOL application for CICS.

   Additionally, if your COBOL application does not make any COBOL MQI call , do not link `libmqmz_r` with `cicsmkcobol`.

   This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call WebSphere MQ on UNIX systems.

   Note: Run `cicsmkcobol` only when you install one of the following:

   - New version or release of Micro Focus COBOL
   - New version or release of CICS for HP-UX
   - New version or release of any supported database product (for COBOL transactions only)
   - New version or release of WebSphere MQ

2. Export the following environment variable:
   ```
   COBCPY=/usr/mqm/inc  export COBCPY
   ```
3. Translate, compile, and link the program by typing:
   ```
   cicstcl -l COBOL -e <yourprog>.ccp
   ```

Address Space models supported by WebSphere MQ for HP-UX on IA64 (IPF)

HP-UX supports two Address Space models:

- MGAS - Mostly Global Address space (this is the default and is used by WebSphere MQ)
- MPAS - Mostly Private Address space

The MPAS model is new on the HP-UX IA64 platform.

For many types of application the Address Space model used has little apparent effect, however there are implications for applications which use System V Shared Memory. In particular, applications using the MPAS model cannot attach System V Shared Memory created by an application which is configured to use the MGAS model when the shared memory is marked as being available to 32-bit applications.

WebSphere MQ is built using the MGAS Address Space model which means that :

- Applications built using the MGAS model can connect to WebSphere MQ.
- Applications built using the MPAS model cannot connect to WebSphere MQ if they connect using the server libraries (libmqm and similar).
- Applications built using the MPAS model can connect to WebSphere MQ if they connect using the client libraries (libmqic and similar), however facilities such as WebSphere MQ trace are, by default, not available.

If you have an application that is built as MPAS you can change it to use the MGAS model using the following command:

```
chatr +as default program-name
```

If you have a requirement to use MPAS applications with WebSphere MQ, you can configure WebSphere MQ to allow both MGAS and MPAS applications to connect by setting the variable AMQ_64BIT_SUBPOOL_ONLY; issue the command

```
export AMQ_64BIT_SUBPOOL_ONLY=1
```

in the environment before starting the queue manager. This allows MPAS applications to connect to the queue manager but it prevents 32-bit applications connecting to the queue manager using the server libraries.

By default COBOL applications built using Micro Focus Server Express are built using the MPAS model. In order to use Micro Focus COBOL applications and WebSphere MQ on the IA64 (IPF) platform you must change the addressing model of your compiled COBOL applications to MGAS or set the AMQ_64BIT_SUBPOOL_ONLY environment variable as described above. More details on the MGAS and MPAS address space models can be found in the HP-UX documentation.

Building your application on Linux

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for Linux applications to run.

C, and C++ are supported. For information about preparing your C++ programs, see WebSphere MQ Using C++.

Preparing C programs

Precompiled C programs are supplied in the `/opt/mqm/samp/bin` directory. To build a sample from source code, use the **gcc** compiler.

Work in your normal environment. Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. To build a sample from source code, use the **gcc** compiler. For further information on programming 64 bit applications see Chapter 10, "Coding standards on 64 bit platforms," on page 579.

Building 31-bit applications

The following section contains examples of the commands used to build 31-bit programs in various environments.

C client application, 31-bit, non-threaded

```
gcc -m31  -o amqsputc_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqic
```

C client application, 31-bit, threaded

```
gcc -m31  -o amqsputc_32_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqic_r -lpthread
```

C server application, 31-bit, non-threaded

```
gcc -m31  -o amqsput_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqm
```

C server application, 31-bit, threaded

```
gcc -m31  -o amqsput_32_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqm_r -lpthread
```

C++ client application, 31-bit, non-threaded

```
g++ -m31 -fsigned-char  -o imqsputc_32 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -limqc23gl
-limqb23gl -lmqic
```

C++ client application, 31-bit, threaded

```
g++ -m31 -fsigned-char  -o imqsputc_32_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -limqc23gl_r
-limqb23gl_r -lmqic_r -lpthread
```

C++ server application, 31-bit, non-threaded

```
g++ -m31 -fsigned-char  -o imqsput_32 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -limqs23gl
-limqb23gl -lmqm
```

C++ server application, 31-bit, threaded

```
g++ -m31 -fsigned-char  -o imqsput_32_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -limqs23gl_r
-limqb23gl_r -lmqm_r -lpthread
```

C client exit, 31-bit, non-threaded

```
gcc -m31 -shared -fPIC  -o /var/mqm/exits/cliexit_32 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqic
```

C client exit, 31-bit, threaded

```
gcc -m31 -shared -fPIC  -o /var/mqm/exits/cliexit_32_r cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqic_r -lpthread
```

C server exit, 31-bit, non-threaded

```
gcc -m31 -shared -fPIC  -o /var/mqm/exits/srvexit_32 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqm
```

C server exit, 31-bit, threaded

```
gcc -m31 -shared -fPIC  -o /var/mqm/exits/srvexit_32_r srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqm_r -lpthread
```

Building 32-bit applications

The following section contains examples of the commands used to build 32-bit programs in various environments.

C client application, 32-bit, non-threaded

```
gcc -m32 -o amqsputc_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqic
```

C client application, 32-bit, threaded

```
gcc -m32 -o amqsputc_32_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqic_r -lpthread
```

C server application, 32-bit, non-threaded

```
gcc -m32 -o amqsput_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqm
```

C server application, 32-bit, threaded

```
gcc -m32 -o amqsput_32_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqm_r -lpthread
```

C++ client application, 32-bit, non-threaded

```
g++ -m32 -fsigned-char -o imqsputc_32 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib  -Wl,-rpath=/usr/lib
-limqc23gl -limqb23gl -lmqic
```

C++ client application, 32-bit, threaded

```
g++ -m32 -fsigned-char -o imqsputc_32_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib  -Wl,-rpath=/usr/lib
-limqc23gl_r -limqb23gl_r -lmqic_r -lpthread
```

C++ server application, 32-bit, non-threaded

```
g++ -m32 -fsigned-char -o imqsput_32 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-limqs23gl -limqb23gl -lmqm
```

C++ server application, 32-bit, threaded

```
g++ -m32 -fsigned-char -o imqsput_32_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-limqs23gl_r -limqb23gl_r -lmqm_r -lpthread
```

C client exit, 32-bit, non-threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/cliexit_32 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqic
```

C client exit, 32-bit, threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/cliexit_32_r cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
-lmqic_r -lpthread
```

C server exit, 32-bit, non-threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/srvexit_32 srvexit.c -I/opt/mqm/inc
-L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib -lmqm
```

C server exit, 32-bit, threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/srvexit_32_r srvexit.c
I/opt/mqm/inc -L/opt/mqm/lib -Wl,-rpath=/opt/mqm/lib -Wl,-rpath=/usr/lib
lmqm_r -lpthread
```

Building 64-bit applications

The following section contains examples of the commands used to build 64-bit programs in various environments.

C client application, 64-bit, non-threaded

```
gcc -m64  -o amqsputc_64 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib64
-Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64 -lmqic
```

C client application, 64-bit, threaded

```
gcc -m64  -o amqsputc_64_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib64
-Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64 -lmqic_r -lpthread
```

C server application, 64-bit, non-threaded

```
gcc -m64  -o amqsput_64 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib64
-Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64 -lmqm
```

C server application, 64-bit, threaded

```
gcc -m64  -o amqsput_64_r amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib64
-Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64 -lmqm_r -lpthread
```

C++ client application, 64-bit, non-threaded

```
g++ -m64 -fsigned-char -o imqsputc_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64
-limqc23gl -limqb23gl -lmqic
```

C++ client application, 64-bit, threaded

```
g++ -m64 -fsigned-char -o imqsputc_64_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64  -Wl,-rpath=/usr/lib64
-limqc23gl_r -limqb23gl_r -lmqic_r -lpthread
```

C++ server application, 64-bit, non-threaded

```
g++ -m64 -fsigned-char  -o imqsput_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64
-limqs23gl -limqb23gl -lmqm
```

C++ server application, 64-bit, threaded

```
g++ -m64 -fsigned-char  -o imqsput_64_r imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64  -Wl,-rpath=/usr/lib64
-limqs23gl_r -limqb23gl_r -lmqm_r -lpthread
```

C client exit, 64-bit, non-threaded

```
gcc -m64 -shared -fPIC  -o /var/mqm/exits64/cliexit_64 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
-Wl,-rpath=/usr/lib64 -lmqic
```

C client exit, 64-bit, threaded

```
gcc -m64 -shared -fPIC  -o /var/mqm/exits64/cliexit_64_r cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
-Wl,-rpath=/usr/lib64 -lmqic_r -lpthread
```

C server exit, 64-bit, non-threaded

```
gcc -m64 -shared -fPIC  -o /var/mqm/exits64/srvexit_64 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
-Wl,-rpath=/usr/lib64 -lmqm
```

C server exit, 64-bit, threaded

```
gcc -m64 -shared -fPIC  -o /var/mqm/exits64/srvexit_64_r srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64
-Wl,-rpath=/usr/lib64 -lmqm_r -lpthread
```

Linking libraries

The following lists the libraries that you need.

- You need to link your programs with the appropriate library provided by WebSphere MQ.

  In a non-threaded environment, link to one of the following libraries:

Library file	Program/exit type
libmqm.so	Server for C
libmqic.so	Client for C

  In a threaded environment, link to one of the following libraries:

Library file	Program/exit type
libmqm_r.so	Server for C
libmqic_r.so	Client for C

Note:

1. If you are writing an installable service (see the WebSphere MQ System Administration Guide for further information), you need to link to the libmqmzf.so library.
2. If you are producing an application for external coordination by an XA-compliant transaction manager such as IBM TXSeries Encina, or BEA Tuxedo, you need to link to the libmqmxa.so (or libmqmxa64.so if your transaction manager treats the 'long' type as 64-bit) and libmqz.so libraries in a non-threaded application and to the libmqmxa_r.so (or libmqmxa64_r.so) and libmqz_r.so libraries in a threaded application.
3. You must link WebSphere MQ libraries before any other product libraries.

Preparing COBOL programs

Notes to users

1. 32 bit COBOL copy books are installed in the following directory:

   `/opt/mqm/inc/cobcpy32`

   and symbolic links are created in:

   `/opt/mqm/inc`

2. On 64-bit platforms, 64 bit COBOL copy books are installed in the following directory:

   `/opt/mqm/inc/cobcpy64`

3. In the following examples set `COBCPY` to:

   `/opt/mqm/inc/cobcpy32`

for 32 bit applications, and:

```
/opt/mqm/inc/cobcpy64
```

for 64 bit applications.

You need to link your program with one of the following:

Library file	Program/exit type
libmqmcb.so	Server for COBOL
libmqicb.so	Client for COBOL
libmqmcb_r.so	Server for COBOL (threaded application)
libmqicb_r.so	Client for COBOL (threaded application)

Preparing COBOL programs using Micro Focus COBOL

Set environment variables before compiling your program as follows:

```
export COBCPY=<COBCPY>
export LIB=/usr/mqm/lib:$LIB
```

To compile a 32 bit COBOL program, where supported, using Micro Focus COBOL, enter:

```
$ cob32 -xvP amqsput.cbl -L /usr/mqm/lib -lmqmcb Server for COBOL
$ cob32 -xvP amqsput.cbl -L /usr/mqm/lib -lmqicb Client for COBOL
$ cob32 -xtvP amqsput.cbl -L /usr/mqm/lib -lmqmcb_r Threaded Server for COBOL
$ cob32 -xtvP amqsput.cbl -L /usr/mqm/lib -lmqicb_r Threaded Client for COBOL
```

To compile a 64 bit COBOL program using Micro Focus COBOL, enter:

```
$ cob64 -xvP amqsput.cbl -L /usr/mqm/lib64 -lmqmcb Server for COBOL
$ cob64 -xvP amqsput.cbl -L /usr/mqm/lib64 -lmqicb Client for COBOL
$ cob64 -xtvP amqsput.cbl -L /usr/mqm/lib64 -lmqmcb_r Threaded Server for COBOL
$ cob64 -xtvP amqsput.cbl -L /usr/mqm/lib64 -lmqicb_r Threaded Client for COBOL
```

where `amqsput` is a sample program

See the Micro Focus COBOL documentation for a description of the environment variables that you need to be up.

Building your application on i5/OS

The i5/OS publications describe how to build executable applications from the programs that you write, to run with i5/OS on iSeries® or System i™ systems.

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for i5/OS applications to run on i5/OS systems. COBOL, C, C++, Java and RPG programming languages are supported. For information about preparing your C++ programs, see WebSphere MQ Using C++. For information about preparing your Java programs, see WebSphere MQ Using Java.

The tasks that you must perform to create an executable WebSphere MQ for i5/OS application depend on the programming language that the source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the WebSphere MQ for i5/OS data

definition files for the language that you are using. Make yourself familiar with the contents of these files. See Chapter 9, "WebSphere MQ data definition files," on page 575 for a full description.

Preparing C programs

WebSphere MQ for i5/OS supports messages up to 100 MB in size. Application programs written in ILE C, supporting WebSphere MQ messages greater than 16 MB, need to use the *Teraspace* compiler option to allocate sufficient memory for these messages.

For further information on the C compiler options, see the *WebSphere WebSphere Development Studio ILE C/C++ Programmer's Guide*.

To compile a C module, you can use the i5/OS command, CRTCMOD. Make sure that the library containing the include files (QMQM) is in the library list when you perform the compilation.

You must then bind the output of the compiler with the service program using the CRTPGM command.

An example of the command for a nonthreaded environment is:

Table 19. Example of CRTPGM in the nonthreaded environment

Command	Program/exit type
`CRTPGM PGM(pgmname) MODULE(pgmname) BNDSRVPGM(QMQM/LIBMQM)`	Server for C

where *pgmname* is the name of your program.

An example of the command for a threaded environment is:

Table 20. Example of CRTPGM in the threaded environment

Command	Program/exit type
`CRTPGM PGM(pgmname) MODULE(pgmname) BNDSRVPGM(QMQM/LIBMQM_R)`	Server for C

where *pgmname* is the name of your program.

Preparing COBOL programs

WebSphere MQ for i5/OS provides two methods for accessing the MQI from within COBOL programs:

1. A dynamic call interface to programs having the names of the MQI functions, such as MQCONN and MQOPEN. This interface is intended primarily for use with the OPM (Original Program Mode) COBOL compiler, but can also be used with the ILE (Integrated Language Environment®) COBOL compiler. Some functions in WebSphere MQ for i5/OS, such as MQCMIT and MQBACK, are not supported through this interface, which is provided for compatibility with previous releases.
2. A bound procedural call interface provided by service programs. This provides access to all the MQI functions in WebSphere MQ for i5/OS, support for threaded applications, and potentially better performance than the dynamic call interface. This interface can be used only with the ILE COBOL compiler.

In both cases the standard COBOL CALL syntax is used to access the MQI functions.

The COBOL copy files containing the named constants and structure definitions for use with the MQI are contained in the source physical file QMQM/QCBLLESRC.

The COBOL copy files use the single quotation mark character (') as the string delimiter. The i5/OS COBOL compilers assume that the delimiter is the double quote("). To prevent the compilers generating warning messages, specify OPTION(*APOST) on the commands CRTCBLPGM, CRTBNDCBL, or CRTCBLMOD.

To make the compiler accept the single quotation mark character (') as the string delimiter in the COBOL copy files, use the compiler option \APOST.

Using the dynamic call interface

- The QMQM library must be in your library list when you compile and when you run COBOL programs using the MQI dynamic call interface.
- Use the CRTCBLPGM command to invoke the OPM COBOL compiler.
- Use either the CRTBNDCBL command or the two separate commands CRTCBLMOD and CRTPGM to invoke the ILE COBOL compiler.

Using the bound procedure call interface

- First create a module using the CRTCBLMOD compiler specifying the parameter:

  ```
  LINKLIT(*PRC)
  ```

- Then use the CRTPGM command to create the program object specifying the parameter:

  for non-threaded applications

  ```
  BNDSRVPGM(QMQM/AMQ0STUB)
  ```

  for threaded applications

  ```
  BNDSRVPGM(QMQM/AMQ0STUB_R)
  ```

Note: Except for programs created using the V4R4 ILE COBOL compiler and containing the THREAD(SERIALIZE) option in the PROCESS statement, COBOL programs must not use the threaded WebSphere MQ libraries. Even if a COBOL program has been made thread safe in this manner, be careful when you design the application, because THREAD(SERIALIZE) forces serialization of COBOL procedures at the module level and might have an impact on overall performance.

See the *WebSphere WebSphere Development Studio: ILE COBOL Programmer's Guide* and the *WebSphere WebSphere Development Studio: ILE COBOL Reference* for further information.

For more information on compiling a CICS application, see the *CICS for i5/OS Application Programming Guide*, SC41-5454.

Preparing CICS programs

To create a program that includes EXEC CICS statements and MQI calls, perform these steps:

1. If necessary, prepare maps using the CRTCICSMAP command.

2. Translate the EXEC CICS commands into native language statements. Use the CRTCICSC command for a C program. Use the CRTCICSCBL command for a COBOL program.

   Include `CICSOPT(*NOGEN)` in the CRTCICSC or CRTCICSCBL command. This halts processing to enable you to include the appropriate CICS and WebSphere MQ service programs. This command puts the code, by default, into QTEMP/QACYCICS.
3. Compile the source code using the CRTCMOD command (for a C program) or the CRTCBLMOD command (for a COBOL program).
4. Use CRTPGM to link the compiled code with the appropriate CICS and WebSphere MQ service programs. This creates the executable program.

An example of such code follows (it compiles the shipped CICS sample program):

```
CRTCICSC  OBJ(QTEMP/AMQSCIC0) SRCFILE(/MQSAMP/QCSRC) +
     SRCMBR(AMQSCIC0) OUTPUT(*PRINT) +
     CICSOPT(*SOURCE *NOGEN)
CRTCMOD   MODULE(MQTEST/AMQSCIC0) +
     SRCFILE(QTEMP/QACYCICS) OUTPUT(*PRINT)
CRTPGM   PGM(MQTEST/AMQSCIC0) MODULE(MQTEST/AMQSCIC0) +
     BNDSRVPGM(QMQM/LIBMQIC QCICS/AEGEIPGM)
```

Preparing RPG programs

If you are using WebSphere MQ for i5/OS, you can write your applications in RPG. For more information see “Coding in RPG” on page 85, and refer to the WebSphere MQ for i5/OS Application Programming Reference (ILE/RPG).

SQL programming considerations

If your program contains EXEC SQL statements and MQI calls, perform these steps:

1. Translate the EXEC SQL commands into native language statements. Use the CRTSQLCI command for a C program. Use the CRTSQLCBLI command for a COBOL program.

   Include `OPTION(*NOGEN)` in the CRTSQLCI or CRTSQLCBLI command. This halts processing to enable you to include the appropriate WebSphere MQ service programs. This command puts the code, by default, into QTEMP/QSQLTEMP.
2. Compile the source code using the CRTCMOD command (for a C program) or the CRTCBLMOD command (for a COBOL program).
3. Use CRTPGM to link the compiled code with the appropriate WebSphere MQ service programs. This creates the executable program.

An example of such code follows (it compiles a program, SQLTEST, in library, SQLUSER):

```
CRTSQLCI  OBJ(MQTEST/SQLTEST) SRCFILE(SQLUSER/QCSRC) +
     SRCMBR(SQLTEST) OUTPUT(*PRINT) OPTION(*NOGEN)
CRTCMOD   MODULE(MQTEST/SQLTEST) +
     SRCFILE(QTEMP/QSQLTEMP) OUTPUT(*PRINT)
CRTPGM   PGM(MQTEST/SQLTEST) +
     BNDSRVPGM(QMQM/LIBMQIC)
```

i5/OS programming considerations

If you have compiled programs for releases of WebSphere MQ for i5/OS earlier than V4R4, you will have linked to AMQZSTUB and, possibly, AMQVSTUB. These libraries are provided at this release for compatibility purposes; you do not need to recompile your applications.

These libraries provide support for the default connection handle (**MQHC_DEF_HCONN**). This is no longer provided by the standard V4R4 libraries. However, the libraries provided at this release for compatibility purposes do not support all new features (for example, **MQCONNX**, **MQCMIT**, and **MQBACK**).

QMQM activation group

When creating your program on i5/OS, the QMQM activation group should not be used. The QMQM activation group is for the use of WebSphere MQ only.

Building your application on Solaris

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for Solaris applications to run under Solaris.

COBOL, C, and C++ programming languages are supported. For information about preparing your C++ programs, see WebSphere MQ Using C++.

In addition to coding the MQI calls in your source code, you must add the appropriate include files. Make yourself familiar with the contents of these files. See Chapter 9, “WebSphere MQ data definition files,” on page 575 for a full description.

Solaris applications must be built threaded, regardless of how many threads the application uses. This is because WebSphere MQ creates background threads. Do not use nonthreadsafe functions such as:

- asctime
- ctime
- qmtime
- localtime
- rand
- srand

Use their threadsafe equivalents.

Throughout this chapter the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Preparing C programs

Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. For further information on programming 64 bit applications see Chapter 10, “Coding standards on 64 bit platforms,” on page 579.

If you want to use the programs on a machine that has only the WebSphere MQ client for Solaris installed, compile the programs to link them with the client library (`-lmqic`).

If you use the unsupported compiler /usr/ucb/cc, your application might compile and link successfully. However when you run it, it will fail when it attempts to connect to the queue manager.

Building applications on x86-64

The following section contains examples of the commands used to build programs in various environments on the x86-64 platform.

C client application, 32-bit

```
cc -xarch=386 -mt -o amqsputc_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -lmqic -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C client application, 64-bit

```
cc -xarch=amd64 -mt -o amqsputc_64 amqsput0.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqic -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C server application, 32-bit

```
cc -xarch=386 -mt -o amqsput_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -lmqm -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C server application, 64-bit

```
cc -xarch=amd64 -mt -o amqsput_64 amqsput0.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqm -lmqmcs -lmqmzse
-lsocket -lnsl -ldl
```

C++ client application, 32-bit

```
CC -xarch=386 -mt -o imqsputc_32 imqsput.cpp -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -limqc23as -limqb23as -lmqic -lmqmcs -lmqmzse
-lsocket -lnsl -ldl
```

C++ client application, 64-bit

```
CC -xarch=amd64 -mt -o imqsputc_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -limqc23as -limqb23as
-lmqic -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C++ server application, 32-bit

```
CC -xarch=386 -mt -o imqsput_32 imqsput.cpp -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -limqs23as -limqb23as -lmqm -lmqmcs -lmqmzse
-lsocket -lnsl -ldl
```

C++ server application, 64-bit

```
CC -xarch=amd64 -mt -o imqsput_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -limqs23as -limqb23as -lmqm
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C client exit, 32-bit

```
cc -xarch=386 -mt -G -KPIC -o /var/mqm/exits/cliexit_32 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib -R/usr/lib/32 -lmqic -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C client exit, 64-bit

```
cc -xarch=amd64 -mt -G -KPIC -o /var/mqm/exits64/cliexit_64 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqic
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C server exit, 32-bit

```
cc -xarch=386 -mt -G -KPIC -o /var/mqm/exits/srvexit_32 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib -R/usr/lib/32 -lmqm -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C server exit, 64-bit

```
cc -xarch=amd64 -mt -G -KPIC -o /var/mqm/exits64/srvexit_64 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqm
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

Building applications on SPARC

The following section contains examples of the commands used to build programs in various environments on the SPARC platform.

C client application, 32-bit

```
cc -xarch=v8plus -mt -o amqsputc_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -lmqic -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C client application, 64-bit

```
cc -xarch=v9 -mt -o amqsputc_64 amqsput0.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqic -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C server application, 32-bit

```
cc -xarch=v8plus -mt -o amqsput_32 amqsput0.c -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -lmqm -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C server application, 64-bit

```
cc -xarch=v9 -mt -o amqsput_64 amqsput0.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqm -lmqmcs -lmqmzse
-lsocket -lnsl -ldl
```

C++ client application, 32-bit

```
CC -xarch=v8plus -mt -o imqsputc_32 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib -R/opt/mqm/lib -R/usr/lib/32 -limqc23as -limqb23as -lmqic
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C++ client application, 64-bit

```
CC -xarch=v9 -mt -o imqsputc_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -limqc23as -limqb23as
-lmqic -lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C++ server application, 32-bit

```
CC -xarch=v8plus -mt -o imqsput_32 imqsput.cpp -I/opt/mqm/inc -L/opt/mqm/lib
-R/opt/mqm/lib -R/usr/lib/32 -limqs23as -limqb23as -lmqm -lmqmcs -lmqmzse
-lsocket -lnsl -ldl
```

C++ server application, 64-bit

```
CC -xarch=v9 -mt -o imqsput_64 imqsput.cpp -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -limqs23as -limqb23as -lmqm
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C client exit, 32-bit

```
cc -xarch=v8plus -mt -G -KPIC -o /var/mqm/exits/cliexit_32 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib -R/usr/lib/32 -lmqic -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C client exit, 64-bit

```
cc -xarch=v9 -mt -G -KPIC -o /var/mqm/exits64/cliexit_64 cliexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqic
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

C server exit, 32-bit

```
cc -xarch=v8plus -mt -G -KPIC -o /var/mqm/exits/srvexit_32 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib -R/opt/mqm/lib -R/usr/lib/32 -lmqm -lmqmcs
-lmqmzse -lsocket -lnsl -ldl
```

C server exit, 64-bit

```
cc -xarch=v9 -mt -G -KPIC -o /var/mqm/exits64/srvexit_64 srvexit.c
-I/opt/mqm/inc -L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqm
-lmqmcs -lmqmzse -lsocket -lnsl -ldl
```

Linking libraries

You must link with the WebSphere MQ libraries that are appropriate for your application type:

Program/exit type	Library files
Server for C	libmqm.so
Client for C	libmqic.so

Note:

1. If you are writing an installable service (see the WebSphere MQ System Administration Guide for further information), link to the libmqmzf.so library.
2. If you are producing an application for external coordination by an XA-compliant transaction manager such as IBM TXSeries Encina, or BEA Tuxedo, you need to link to the libmqmxa.so (or libmqmxa64.so if your transaction manager treats the 'long' type as 64-bit) and libmqz.so libraries.
3. You must link WebSphere MQ libraries before any other product libraries.

Preparing COBOL programs

Notes to users

1. 32 bit COBOL copy books are installed in the following directory:
   `/opt/mqm/inc/cobcpy32`

   and symbolic links are created in:
   `/opt/mqm/inc`
2. 64 bit COBOL copy books are installed in the following directory:
   `/opt/mqm/inc/cobcpy64`
3. In the following examples set `COBCPY` to:
   `/opt/mqm/inc/cobcpy32`

   for 32 bit applications, and:
   `/opt/mqm/inc/cobcpy64`

   for 64 bit applications.

Compile the programs using Micro Focus compiler. The copy files that declare the structures are in /opt/mqm/inc:

```
$ export LIB=/opt/mqm/lib:$LIB
$ export COBCPY="<COBCPY>"
```

Compiling 32 bit programs:

- $ cob32 -xv amqs0put0.cbl -L /opt/mqm/lib -lmqmcb
  Server for COBOL
- $ cob32 -xv amqs0put0.cbl -L /opt/mqm/lib -lmqicb
  Client for COBOL
- $ cob32 -xtv amqs0put0.cbl -L /opt/mqm/lib -lmqmcb_r

Threaded Server for COBOL

- $ cob32 -xtv amqs0put0.cbl -L /opt/mqm/lib -lmqicb_r
  Threaded Client for COBOL

Compiling 64-bit programs:

- $ cob64 -xv amqs0put0.cbl -L /opt/mqm/lib64 -lmqmcb
  Server for COBOL
- $ cob64 -xv amqs0put0.cbl -L /opt/mqm/lib64 -lmqicb
  Client for COBOL
- $ cob64 -xtv amqs0put0.cbl -L /opt/mqm/lib64 -lmqmcb_r
  Threaded Server for COBOL
- $ cob64 -xtv amqs0put0.cbl -L /opt/mqm/lib64 -lmqicb_r
  Threaded Client for COBOL

where amqs0put0.cbl is a sample program

You need to link your program with one of the following:

- libmqmcb.sl
  Server for COBOL
- libmqicb.sl
  Client for COBOL
- libmqmcb_r.sl
  Threaded Server applications
- libmqicb_r.sl
  Threaded Client applications

Preparing CICS programs

An XA switch module is provided to enable you to link CICS with WebSphere MQ:

Table 21. Essential code for CICS applications (Solaris)

Description	C (source)	C (exec)
XA initialization routine	amqzscix.c	amqzsc - TXSeries for Solaris

Always link your transactions with the thread safe WebSphere MQ library libmqm_so.

You can find more information about supporting CICS transactions in the WebSphere MQ System Administration Guide.

TXSeries CICS support

WebSphere MQ for Solaris supports TXSeries CICS using the XA interface.

Preparing CICS COBOL programs using Micro Focus COBOL:

To use Micro Focus COBOL, follow these steps:

1. Add the WebSphere MQ COBOL run-time library module to the run-time library using the following command:

```
cicsmkcobol -L/usr/lib/dce -L/opt/mqm/lib  \
            /opt/mqm/lib/libmqmcbrt.o -lmqm
```

Note: With `cicsmkcobol`, WebSphere MQ does not allow you to make MQI calls in the C programming language from your COBOL application.

If your existing applications have any such calls, you are strongly recommended to move these functions from the COBOL applications to your own library, for example, `myMQ.so`. After you have done this, do not include the WebSphere MQ library `libmqmcbrt.o` when building the COBOL application for CICS.

Additionally, if your COBOL application does not make any COBOL MQI call , do not link `libmqmz_r` with `cicsmkcobol`.

This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call WebSphere MQ on UNIX systems.

Note: Run `cicsmkcobol` only when you install one of the following:

- New version or release of Micro Focus COBOL
- New version or release of TXSeries for Solaris
- New version or release of any supported database product (for COBOL transactions only)
- New version or release of WebSphere MQ

2. Export the following environment variable:
   ```
   COBCPY=/usr/mqm/inc  export COBCPY
   ```
3. Translate, compile, and link the program by typing:
   ```
   cicstcl -l COBOL -e <yourprog>.ccp
   ```

Preparing CICS C programs:

Build CICS C programs using the standard CICS facilities:

1. Export ***one*** of the following environment variables:
   - LDFLAGS = "-L/usr/mqm/lib -lmqm_r" export LDFLAGS
   - USERLIB = "-L/usr/mqm/lib -lmqm_r" export USERLIB
2. Translate, compile, and link the program by typing:
   ```
   cicstcl -l C amqscic0.ccs
   ```

CICS C sample transaction:

Sample C source for a CICS WebSphere MQ transaction is provided by AMQSCIC0.CCS. The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures are sent to the queue SYSTEM.SAMPLE.CICS.DLQ. Use the sample MQSC script AMQSCIC0.TST to create these queues and sample input queues.

Building your application on Windows systems

The Windows systems publications describe how to build executable applications from the programs that you write.

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for Windows applications to run under Windows systems. ActiveX, C, C++, COBOL, and Visual Basic programming languages are supported. For information about preparing your ActiveX programs, see WebSphere MQ Using the Component Object Model Interface. For information about preparing your C++ programs, see WebSphere MQ Using C++.

The tasks that you must perform to create an executable application using WebSphere MQ for Windows vary with the programming language that your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the WebSphere MQ for Windows include files for the language that you are using. Make yourself familiar with the contents of these files. See Chapter 9, "WebSphere MQ data definition files," on page 575 for a full description.

Building 64-bit applications on Windows

Both 32-bit and 64-bit applications are supported on WebSphere MQ for Windows, Version 7.0. The WebSphere MQ executable and library files are supplied in both 32-bit and 64-bit forms, use the appropriate version depending on the application you are working with.

Executable files and libraries

Both 32-bit and 64-bit versions of the WebSphere MQ libraries are supplied in the following locations:

Table 22. Location of WebSphere MQ libraries

Library version	Directory containing library files
32-bit	install_location\Tools\Lib
64-bit	install_location\Tools\Lib64

32-bit applications continue to work normally after migration. The 32-bit files exist in the same directory as in previous versions of the product.

If you want to create 64-bit version you must ensure that your environment is configured to use the library files in install_location\Tools\Lib64. Ensure that the LIB environment variable is not set to look in the folder containing the 32-bit libraries.

Preparing C programs

Work in your usual Windows environment; WebSphere MQ for Windows requires nothing special.

For further information on programming 64-bit applications see Chapter 10, "Coding standards on 64 bit platforms," on page 579.

- Link your programs with the appropriate libraries provided by WebSphere MQ:

Library file	Program/exit type
install_location\Tools\Lib\mqm.lib	server for 32-bit C
install_location\Tools\Lib\mqic.lib	client for 32-bit C

Library file	Program/exit type
install_location\Tools\Lib\mqicxa.lib	client for 32-bit C with transaction co-ordination
install_location\Tools\Lib64\mqm.lib	server for 64-bit C
install_location\Tools\Lib64\mqic.lib	client for 64-bit C
install_location\Tools\Lib64\mqicxa.lib	client for 64-bit C with transaction co-ordination

The following command gives an example of compiling the sample program amqsget0 (using the Microsoft® Visual C++ compiler).

For 32-bit applications:

```
cl -MD amqsget0.c /Fe amqsget.exe install_location/Tools/Lib/mqm.lib
```

For 64-bit applications:

```
cl -MD amqsget0.c /Fe amqsget.exe install_location/Tools/Lib64/mqm.lib
```

Note:

- If you are writing an installable service (see the WebSphere MQ System Administration Guide for further information), you need to link to the mqmzf.lib library.
- If you are producing an application for external coordination by an XA-compliant transaction manager such as IBM TXSeries Encina, or BEA Tuxedo, you need to link to the mqmxa.lib or mqmxa.lib library.
- If you are writing a CICS exit, link to the mqmcics4.lib library.
- You must link WebSphere MQ libraries before any other product libraries.

- The DLLs must be in the path (PATH) that you have specified.
- If you use lowercase characters whenever possible, you can move from WebSphere MQ for Windows to WebSphere MQ on UNIX systems, where use of lowercase is necessary.

Preparing CICS and Transaction Server programs

Sample C source for a CICS WebSphere MQ transaction is provided by AMQSCIC0.CCS. You build it using the standard CICS facilities. For example, for TXSeries for Windows 2000:

1. Set the environment variable (enter the following on one line):

   ```
   set CICS_IBMC_FLAGS=-IC:\Program Files\IBM\WebSphere MQ\Tools\C\Include;
   %CICS_IBMC_FLAGS%
   ```

2. Set the USERLIB environment variable:

   ```
   set USERLIB=MQM.LIB;%USERLIB%
   ```

3. Translate, compile, and link the sample program:

   ```
   cicstcl -l IBMC amqscic0.ccs
   ```

This is described in the *Transaction Server for Windows NT Application Programming Guide (CICS) V4*.

You can find more information about supporting CICS transactions in the WebSphere MQ System Administration Guide.

Preparing COBOL programs

Notes to users

1. The 32-bit COBOL copy books are installed in the following directory:
   `C:\Program Files\IBM\WebSphere MQ\Tools\cobol\CopyBook`
2. The 64-bit COBOL copy books are installed in the following directory:
   `C:\Program Files\IBM\WebSphere MQ\Tools\cobol\CopyBook64`
3. In the following examples set `CopyBook` to:
   `CopyBook`

   for 32-bit applications, and:
   `CopyBook64`

   for 64-bit applications.

To prepare COBOL programs on Windows systems, link your program to one of the following libraries provided by WebSphere MQ:

Library file	Program or exit type
install_location\Tools\Lib\mqmcbb	32-bit server for IBM COBOL
install_location\Tools\Lib\mqmcb	32-bit server for Micro Focus COBOL
install_location\Tools\Lib\mqiccbb	32-bit client for IBM COBOL
install_location\Tools\Lib\mqiccb	32-bit client for Micro Focus COBOL
install_location\Tools\Lib64\mqmcbb	64-bit server for IBM COBOL
install_location\Tools\Lib64\mqmcb	64-bit server for Micro Focus COBOL
install_location\Tools\Lib64\mqiccbb	64-bit client for IBM COBOL
install_location\Tools\Lib64\mqiccb	64-bit client for Micro Focus COBOL

When you are running a program in the MQI client environment, ensure that the DOSCALLS library appears before any COBOL or WebSphere MQ library.

You can use the IBM COBOL Set compiler or Micro Focus COBOL compiler depending on the program:

- Programs beginning `amqi` are suitable for the IBM COBOL Set compiler,
- Programs beginning `amqm` are suitable for the Micro Focus COBOL compiler, and
- Programs beginning `amq0` are suitable for either compiler.

IBM and Micro Focus COBOL

Relink any existing 32-bit WebSphere MQ Micro Focus COBOL programs using either mqmcb.lib or mqiccb.lib, rather than the mqmcbb and mqiccbb libraries.

To compile, for example, the sample program amq0put0, using IBM VisualAge COBOL:

1. Set the SYSLIB environment variable to include the path to the WebSphere MQ VisualAge COBOL copybooks (enter the following on one line):

```
set SYSLIB=<drive>:\Program Files\IBM\WebSphere MQ\
Tools\Cobol\Copybook\VAcobol;%SYSLIB%
```

2. For use on the WebSphere MQ server:

```
cob2 amq0put0.cbl -qlib "<drive>:\Program Files\IBM\WebSphere MQ\
Tools\Lib\mqmcbb.lib"
```

3. For use on the WebSphere MQ client:

```
cob2 amq0put0.cbl -qlib "<drive>:\Program Files\IBM\WebSphere MQ\
Tools\Lib\mqiccbb.lib"
```

Note: Although you must use the compiler option CALLINT(SYSTEM), this is the default for `cob2`.

To compile, for example, the sample program amq0put0, using Micro Focus COBOL:

1. Set the COBCPY environment variable to point to the WebSphere MQ COBOL copybooks (enter the following on one line):

```
set COBCPY=<drive>:\Program Files\IBM\WebSphere MQ\
Tools\Cobol\Copybook
```

2. Compile the program to give you an object file:

```
cobol amq0put0 LITLINK
```

3. Link the object file to the runtime system.
   - Set the LIB environment variable to point to the compiler COBOL libraries.
   - Link the object file for use on the WebSphere MQ server:

     ```
     cbllink amq0put0.obj mqmcb.lib
     ```

   - Or link the object file for use on the WebSphere MQ client:

     ```
     cbllink amq0put0.obj mqiccb.lib
     ```

Preparing CICS and Transaction Server programs

To compile and link a TXSeries for Windows NT®, V5.1 program using IBM VisualAge COBOL:

1. Set the environment variable (enter the following on one line):

```
set CICS_IBMCOB_FLAGS=c:\Program Files\IBM\WebSphere MQ\Tools\
Cobol\Copybook\VAcobol;%CICS_IBMCOB_FLAGS%
```

2. Set the USERLIB environment variable:

```
set USERLIB=MQMCBB.LIB
```

3. Translate, compile, and link your program:

```
cicstcl -l IBMCOB myprog.ccp
```

This is described in the *Transaction Server for Windows NT, V4 Application Programming* Guide.

To compile and link a CICS for Windows V5 program using Micro Focus COBOL:

- Set the INCLUDE variable:

```
set
INCLUDE=<drive>:\<programname>\ibm\websphere\tools\c\include;
        <drive>:\opt\cics\include;%INCLUDE%
```

- Set the COBCPY environment variable:

```
setCOBCPY=<drive>:\<programname>\ibm\websphere\tools\cobol\copybook;
          <drive>:\opt\cics\include
```

- Set the COBOL options:

- set
- COBOPTS=/LITLINK /NOTRUNC

and run the following code:

```
cicstran cicsmq00.ccp
cobol cicsmq00.cbl /LITLINK /NOTRUNC
cbllink -D -Mcicsmq00 -Ocicsmq00.cbmfnt cicsmq00.obj
%CICSLIB%\cicsprCBMFNT.lib user32.lib msvcrt.lib kernel32.lib mqmcb32.lib
```

Preparing Visual Basic programs

Note: 64-bit versions of the Visual Basic module files are not supplied.

To prepare Visual Basic programs on Windows:

1. Create a new project.
2. Add the supplied module file, CMQB.BAS, to the project.
3. Add other supplied module files if you need them:

CMQBB.BAS	MQAI support
CMQCFB.BAS	PCF support
CMQXB.BAS	Channel exits support
CMQPSB.BAS	Publish/subscribe

See "Coding in Visual Basic" on page 86 for information about using the MQCONNXAny call from within Visual Basic.

Call the procedure MQ_SETDEFAULTS before making any MQI calls in the project code. This procedure sets up default structures that the MQI calls require.

Specify whether you are creating a WebSphere MQ server or client, before you compile or run the project, by setting the conditional compilation variable *MqType*. Set *MqType* in a Visual Basic project to 1 for a server or 2 for a client as follows:

1. Select the Project menu.
2. Select *Name* Properties (where *Name* is the name of the current project).
3. Select the Make tab in the dialog box.
4. In the Conditional Compilation Arguments field, enter this for a server:

   ```
   MqType=1
   ```

   or this for a client:

   ```
   MqType=2
   ```

SSPI security exit

WebSphere MQ for Windows supplies a security exit for both the WebSphere MQ client and the WebSphere MQ server. This is a channel-exit program that provides authentication for WebSphere MQ channels by using the Security Services Programming Interface (SSPI). The SSPI provides the integrated security facilities of Windows systems.

The security packages are loaded from either security.dll or secur32.dll. These DLLs are supplied with your operating system.

One-way authentication is provided using NTLM authentication services. Two way authentication is provided using Kerberos authentication services.

The security exit program is supplied in source and object format. You can use the object code as it is, or you can use the source code as a starting point to create your own user-exit programs.

See also "Using the SSPI security exit on Windows systems" on page 452.

Introduction to security exits

A security exit forms a secure connection between two security exit programs, where one program is for the sending message channel agent (MCA), and one is for the receiving MCA.

The program that initiates the secure connection, that is, the first program to get control after the MCA session is established, is known as the *context initiator*. The partner program is known as the *context acceptor*.

The following table shows some of the channel types that are context initiators and their associated context acceptors.

Table 23. Context initiators and their associated context acceptors

Context Initiator	Context Acceptor
MQCHT_CLNTCONN	MQCHT_SVRCONN
MQCHT_RECEIVER	MQCHT_SENDER
MQCHT_CLUSRCVR	MQCHT_CLUSSDR

The security exit program has two entry points:

- **SCY_NTLM**

  This uses NTLM authentication services, which provide one-way authentication. NTLM allows servers to verify the identities of their clients. It does not allow clients to verify a server's identity, or one server to verify the identity of another. NTLM authentication was designed for a network environment in which servers are assumed to be genuine.
- **SCY_KERBEROS**

  This uses Kerberos mutual authentication services. The Kerberos protocol does not assume that servers in a network environment are genuine. Parties at both ends of a network connection can verify the identity of the other party. That is, servers can verify the identity of clients and other servers, and clients can verify the identity of a server.

What the security exit does:

This section describes what the SSPI channel-exit programs do.

The supplied channel-exit programs provide either one-way or two-way (mutual) authentication of a partner system when a session is being established. For a particular channel, each exit program has an associated *principal* (similar to a user ID, see "WebSphere MQ access control and Windows principals" on page 372). A connection between two exit programs is an association between the two principals.

After the underlying session is established, a secure connection between two security exit programs (one for the sending MCA and one for the receiving MCA), is established. The sequence of operations is as follows:

1. Each program is associated with a particular principal, for example as a result of an explicit login operation.
2. The context initiator requests a secure connection with the partner from the security package (for Kerberos, the named partner) and receives a token (called token1). The token is sent, using the underlying session that is already established, to the partner program.
3. The partner program (the context acceptor) passes token1 to the security package, which verifies that the context initiator is authentic. For NTLM, the connection is now established.
4. For the Kerberos-supplied security exit (that is, for mutual authentication), the security package also generates a second token (called token2), which the context acceptor returns to the context initiator by using the underlying session.
5. The context initiator uses token2 to verify that the context acceptor is authentic.
6. At this stage, if both applications are satisfied with the authenticity of the partner's token, the secure (authenticated) connection is established.

WebSphere MQ access control and Windows principals:

The access control that WebSphere MQ provides is based on the user and group. The authentication that Windows provides is based on principals, such as user and servicePrincipalName (SPN). In the case of servicePrincipalName, there might be many of these associated with a single user.

The SSPI security exit uses the relevant Windows principals for authentication. If Windows authentication is successful, the exit passes the user ID that is associated with the Windows principal to WebSphere MQ for access control.

The Windows principals that are relevant for authentication vary, depending on the type of authentication used.

- For NTLM authentication, the Windows principal for Context Initiator is the user ID associated with the process that is running. Because this authentication is one-way, the principal associated with the Context Acceptor is irrelevant.
- For Kerberos authentication, on CLNTCONN channels, the Windows principal is the user ID associated with the process that is running. Otherwise, the Windows principal is the servicePrincipalName that is formed by adding the following prefix to the QueueManagerName.

  ```
  ibmMQSeries/
  ```

Building your application on z/OS

The CICS, IMS, and z/OS publications describe how to build applications that run in these environments.

This chapter describes the additional tasks, and the changes to the standard tasks, that you must perform when building WebSphere MQ for z/OS applications for these environments. COBOL, C, C++, Assembler, and PL/I programming languages are supported. (For information on building C++ applications see WebSphere MQ Using C++.)

The tasks that you must perform to create an executable WebSphere MQ for z/OS application depend on both the programming language that the program is written in, and the environment in which the application will run.

In addition to coding the MQI calls in your program, add the appropriate language statements to include the WebSphere MQ for z/OS data definition file for the language that you are using. Make yourself familiar with the contents of these files. See Chapter 9, "WebSphere MQ data definition files," on page 575 for a full description.

Note

The name **thlqual** is the high-level qualifier of the installation library on z/OS.

This chapter introduces building z/OS applications, under these headings:

- "Preparing your program to run"
- "Dynamically calling the WebSphere MQ stub" on page 377
- "Debugging your programs" on page 382

Preparing your program to run

After you have written the program for your WebSphere MQ application, to create an executable application you have to compile or assemble it, then link-edit the resulting object code with the stub program that WebSphere MQ for z/OS supplies for each environment that it supports.

How you prepare your program depends on both the environment (batch, CICS, IMS(BMP or MPP), or UNIX System services) in which the application runs, and the structure of the data sets on your z/OS installation. The details are described in the following sections.

"Dynamically calling the WebSphere MQ stub" on page 377 describes an alternative method of making MQI calls in your programs so that you do not need to link-edit a WebSphere MQ stub. This method is not available for all languages and environments.

Do not link-edit a higher level of stub program than that of the version of WebSphere MQ for z/OS on which your program is running. For example, a program running on MQSeries for OS/390, V5.2 must not be link-edited with a stub program supplied with WebSphere MQ for z/OS V7.

Building z/OS batch applications

To build an application for WebSphere MQ for z/OS that runs under z/OS batch, create job control language (JCL) that performs these tasks:

1. Compile (or assemble) the program to produce object code. The JCL for your compilation must include SYSLIB statements that make the product data definition files available to the compiler. The data definitions are supplied in the following WebSphere MQ for z/OS libraries:
   - For COBOL, **thlqual**.SCSQCOBC
   - For assembler language, **thlqual**.SCSQMACS
   - For C, **thlqual**.SCSQC370
   - For PL/I, **thlqual**.SCSQPLIC
2. For a C application, prelink the object code created in step 1.
3. Link-edit the object code created in step 1 (or step 2 for a C application) to produce a load module. When you link-edit the code, you must include one of the WebSphere MQ for z/OS batch stub programs (CSQBSTUB or one of the RRS stub programs: CSQBRRSI or CSQBRSTB).

CSQBSTUB
single-phase commit provided by WebSphere MQ for z/OS

CSQBRRSI
two-phase commit provided by RRS using the MQI

CSQBRSTB
two-phase commit provided by RRS directly

Note: If you use CSQBRSTB, you must also link-edit your application with ATRSCSS from SYS1.CSSLIB. Figure 24 and Figure 25 show fragments of JCL to do this. The stubs are language-independent and are supplied in library **thlqual**.SCSQLOAD.

4. Store the load module in an application load library.

```
.
.
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING BATCH STUB
//*
//CSQSTUB   DD  DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
//*
.
.
//SYSIN     DD *
  INCLUDE CSQSTUB(CSQBSTUB)
.
.
/*
```

Figure 24. Fragments of JCL to link-edit the object module in the batch environment, using single-phase commit

```
.
.
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING BATCH STUB
//*
//CSQSTUB   DD  DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
//CSSLIB    DD  DSN=SYS1.CSSLIB,DISP=SHR
//*
.
.
//SYSIN     DD *
  INCLUDE CSQSTUB(CSQBRSTB)
  INCLUDE CSSLIB(ATRSCSS)
.
.
/*
```

Figure 25. Fragments of JCL to link-edit the object module in the batch environment, using two-phase commit

To run a batch or RRS program, you must include the libraries **thlqual**.SCSQAUTH and **thlqual**.SCSQLOAD in the STEPLIB or JOBLIB data set concatenation.

To run a TSO program, you must include the libraries **thlqual**.SCSQAUTH and **thlqual**.SCSQLOAD in the STEPLIB used by the TSO session.

To run an OpenEdition batch program from the OpenEdition shell, add the libraries **thlqual**.SCSQAUTH and **thlqual**.SCSQLOAD to the STEPLIB specification in your $HOME/.profile like this:

```
STEPLIB=thlqual.SCSQAUTH:thlqual.SCSQLOAD
export STEPLIB
```

Building CICS applications

To build an application for WebSphere MQ for z/OS that runs under CICS, you must:

- Translate the CICS commands in your program into the language in which the rest of your program is written
- Compile or assemble the output from the translator to produce object code
- Link-edit the object code to create a load module

CICS provides a procedure to execute these steps in sequence for each of the programming languages it supports.

- For CICS Transaction Server for OS/390, the *CICS Transaction Server for OS/390 System Definition Guide* describes how to use these procedures and the *CICS/ESA Application Programming Guide* gives more information on the translation process.

You must include:

- In the SYSLIB statement of the compilation (or assembly) stage, statements that make the product data definition files available to the compiler. The data definitions are supplied in the following WebSphere MQ for z/OS libraries:
  - For COBOL, **thlqual**.SCSQCOBC
  - For assembler language, **thlqual**.SCSQMACS
  - For C, **thlqual**.SCSQC370
  - For PL/I, **thlqual**.SCSQPLIC
- In your link-edit JCL, the WebSphere MQ for z/OS CICS stub program (CSQCSTUB). Figure 26 shows fragments of JCL code to do this. The stub is language-independent and is supplied in library **thlqual**.SCSQLOAD.

When you have completed these steps, store the load module in an application load library and define the program to CICS in the usual way.

```
.
.
.
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING CICS STUB
//*
//CSQSTUB   DD  DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
//*
.
.
.
//LKED.SYSIN DD *
  INCLUDE CSQSTUB(CSQCSTUB)
.
.
.
/*
```

Figure 26. Fragments of JCL to link-edit the object module in the CICS environment

Before you run a CICS program, your system administrator must define it to CICS as a WebSphere MQ program and transaction, You can then run it in the usual way.

Building IMS (BMP or MPP) applications

If you are building batch DL/I programs, see "Building z/OS batch applications" on page 373. To build other applications that run under IMS (either as a BMP or an MPP), create JCL that performs these tasks:

1. Compile (or assemble) the program to produce object code. The JCL for your compilation must include SYSLIB statements that make the product data

definition files available to the compiler. The data definitions are supplied in the following WebSphere MQ for z/OS libraries:

- For COBOL, **thlqual**.SCSQCOBC
- For assembler language, **thlqual**.SCSQMACS
- For C, **thlqual**.SCSQC370
- For PL/I, **thlqual**.SCSQPLIC

2. For a C application, prelink the object module created in step 1 on page 375.
3. Link-edit the object code created in step 1 on page 375 (or step 2 for a C/370 application) to produce a load module:
   a. Include the IMS language interface module (DFSLI000).
   b. Include the WebSphere MQ for z/OS IMS stub program (CSQQSTUB). Figure 27 shows fragments of JCL to do this. The stub is language independent and is supplied in library **thlqual**.SCSQLOAD.

      Note: If you are using COBOL, select the NODYNAM compiler option to enable the linkage editor to resolve references to CSQQSTUB unless you intend to use dynamic linking as described in "Dynamically calling the WebSphere MQ stub" on page 377.
4. Store the load module in an application load library.

```
.
.
.
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING IMS STUB
//*
//CSQSTUB   DD  DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
//*
.
.
.
//LKED.SYSIN DD *
  INCLUDE CSQSTUB(CSQQSTUB)
.
.
.
/*
```

Figure 27. Fragments of JCL to link-edit the object module in the IMS environment

Before you run an IMS program, your system administrator must define it to IMS as a WebSphere MQ program and transaction: you can then run it in the usual way.

Building z/OS UNIX System Services applications

To build a C application for WebSphere MQ for z/OS that runs under UNIX System Services, compile and link your application as follows:

```
cc -o mqsamp -I "//'thlqual.SCSQC370'" mqsamp.c
"//'thlqual.SCSQLOAD(CSQBSTUB)'"
```

where **thlqual** is the high-level qualifier used by your installation.

To run the C program, you need to add the following to your `.profile` file; this should be in your root directory:

```
STEPLIB=thlqual.SCSQANLE:thlqual.SCSQAUTH: STEPLIB
```

Note that you need to exit from OpenEdition, and enter OpenEdition again, for the change to be recognized.

If you want to run multiple shells, add the word export at the beginning of the line, that is:

```
export STEPLIB=thlqual.SCSQANLE:thlqual.SCSQAUTH: STEPLIB
```

Once this completes successfully you can link the CSQBSTUB and issue WebSphere MQ calls.

"Dynamically calling the WebSphere MQ stub" describes an alternative method of making MQI calls in your programs so that you do not need to link-edit a WebSphere MQ stub. This method is not available for all languages and environments.

Do not link-edit a higher level of stub program than that of the version of WebSphere MQ for z/OS on which your program is running. For example, a program running on MQSeries for OS/390, V5.2 must not be link-edited with a stub program supplied with WebSphere MQ for z/OS, V5.3.

Dynamically calling the WebSphere MQ stub

Instead of link-editing the WebSphere MQ stub program with your object code, you can dynamically call the stub from within your program.

You can do this in the batch, IMS, and CICS environments. This facility is not supported by programs using PL/I in the CICS environment. and it is not supported in the RRS environment. If your application program uses RRS to coordinate updates, see "RRS Considerations" on page 381.

However, this method:

- Increases the complexity of your programs
- Increases the storage required by your programs at execution time
- Reduces the performance of your programs
- Means that you cannot use the same programs in other environments

If you call the stub dynamically, the appropriate stub program and its aliases must be available at execution time. To ensure this, include the WebSphere MQ for z/OS data set SCSQLOAD:

For batch and IMS	In the STEPLIB concatenation of the JCL
For CICS	In the CICS DFHRPL concatenation

For IMS, ensure that the library containing the dynamic stub (built as described in the information about installing the IMS adapter in the WebSphere MQ for z/OS System Setup Guide) is ahead of the data set SCSQLOAD in the STEPLIB concatenation of the region JCL.

Use the names shown in Table 24 when you call the stub dynamically. In PL/I, only declare the call names used in your program.

Table 24. Call names for dynamic linking

MQI call	**Dynamic call name**		
	Batch (non-RRS)	**CICS**	**IMS**
MQBACK	CSQBBACK	not supported	not supported
MQBUFMH	CSQBBFMH	CSQCBFMH	MQBUFMH

Table 24. Call names for dynamic linking (continued)

MQI call	Dynamic call name		
MQCB	CSQBCB	CSQCCB	MQCB
MQCLOSE	CSQBCLOS	CSQCCLOS	MQCLOSE
MQCMIT	CSQBCOMM	not supported	not supported
MQCONN	CSQBCONN	CSQCCONN	MQCONN
MQCONNX	CSQBCONX	CSQCCONX	MQCONNX
MQCRTMH	CSQBCRMH	CSQCCRMH	MQCRTMH
MQCTL	CSQBCTL	CSQCCTL	MQCTL
MQDISC	CSQBDISC	CSQCDISC	MQDISC
MQDLTMH	CSQBDLMH	CSQCDLMH	MQDLTMH
MQDLTMP	CSQBDLMP	CSQCDLMP	MQDLTMP
MQGET	CSQBGET	CSQCGET	MQGET
MQINQ	CSQBINQ	CSQCINQ	MQINQ
MQINQMP	CSQBINMP	CSQCINMP	MQINQMP
MQMHBUF	CSQBMHBF	CSQCMHBF	MQMHBUF
MQOPEN	CSQBOPEN	CSQCOPEN	MQOPEN
MQPUT	CSQBPUT	CSQCPUT	MQPUT
MQPUT1	CSQBPUT1	CSQCPUT1	MQPUT1
MQSET	CSQBSET	CSQCSET	MQSET
MQSETMP	CSQBSTMP	CSQCSTMP	MQSETMP
MQSTAT	CSQBSTAT	CSQCSTAT	MQSTAT
MQSUB	CSQBSUB	CSQCSUB	MQSUB
MQSUBRQ	CSQBSBRQ	CSQCSBRQ	MQSUBRQ

For examples of how to use this technique, see the following figures:

Batch and COBOL	Figure 28 on page 379
CICS and COBOL	Figure 29 on page 379
IMS and COBOL	Figure 30 on page 379
Batch and assembler	Figure 31 on page 380
CICS and assembler	Figure 32 on page 380
IMS and assembler	Figure 33 on page 380
Batch and C	Figure 34 on page 380
CICS and C	Figure 35 on page 380
IMS and C	Figure 36 on page 381
Batch and PL/I	Figure 37 on page 381
IMS and PL/I	Figure 38 on page 381

```
...
       WORKING-STORAGE SECTION.
...
          05 WS-MQOPEN                    PIC X(8) VALUE 'CSQBOPEN'.
...
       PROCEDURE DIVISION.
...
           CALL WS-MQOPEN WS-HCONN
                          MQOD
                          WS-OPTIONS
                          WS-HOBJ
                          WS-COMPCODE
                          WS-REASON.
...
```

Figure 28. Dynamic linking using COBOL in the batch environment

```
...
       WORKING-STORAGE SECTION.
...
          05 WS-MQOPEN                    PIC X(8) VALUE 'CSQCOPEN'.
...
       PROCEDURE DIVISION.
...
           CALL WS-MQOPEN WS-HCONN
                          MQOD
                          WS-OPTIONS
                          WS-HOBJ
                          WS-COMPCODE
                          WS-REASON.
...
```

Figure 29. Dynamic linking using COBOL in the CICS environment

```
...
       WORKING-STORAGE SECTION.
...
          05 WS-MQOPEN                    PIC X(8) VALUE 'MQOPEN'.
...
       PROCEDURE DIVISION.
...
           CALL WS-MQOPEN WS-HCONN
                          MQOD
                          WS-OPTIONS
                          WS-HOBJ
                          WS-COMPCODE
                          WS-REASON.
...
      * ------------------------------------------------------------ *
      *
      *    If the compile option 'DYNAM' is specified
      *    then you may code the MQ calls as follows
      *
      * ------------------------------------------------------------ *
...
           CALL 'MQOPEN'  WS-HCONN
                          MQOD
                          WS-OPTIONS
                          WS-HOBJ
                          WS-COMPCODE
                          WS-REASON.
...
```

Figure 30. Dynamic linking using COBOL in the IMS environment

```
...
         LOAD   EP=CSQBOPEN
...
         CALL (15),(HCONN,MQOD,OPTIONS,HOBJ,COMPCODE,REASON),VL
...
         DELETE EP=CSQBOPEN
...
```

Figure 31. Dynamic linking using assembler language in the batch environment

```
...
         EXEC CICS LOAD PROGRAM('CSQCOPEN') ENTRY(R15)
...
         CALL (15),(HCONN,MQOD,OPTIONS,HOBJ,COMPCODE,REASON),VL
...
         EXEC CICS RELEASE PROGRAM('CSQCOPEN')
...
```

Figure 32. Dynamic linking using assembler language in the CICS environment

```
...
         LOAD   EP=MQOPEN
...
         CALL (15),(HCONN,MQOD,OPTIONS,HOBJ,COMPCODE,REASON),VL
...
         DELETE EP=MQOPEN
...
```

Figure 33. Dynamic linking using assembler language in the IMS environment

```
...
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
...
main()
{
CALL_ME * csqbopen;
...
csqbopen = (CALL_ME *) fetch("CSQBOPEN");
(*csqbopen)(Hconn,&ObjDesc,Options,&Hobj,&CompCode,&Reason);
...
```

Figure 34. Dynamic linking using C language in the batch environment

```
...
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
...
main()
{
CALL_ME * csqcopen;
...
  EXEC CICS LOAD PROGRAM("CSQCOPEN") ENTRY(csqcopen);
(*csqcopen)(Hconn,&ObjDesc,Options,&Hobj,&CompCode,&Reason);
...
```

Figure 35. Dynamic linking using C language in the CICS environment

```
...
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
...
main()
{
CALL_ME * mqopen;
...
mqopen = (CALL_ME *) fetch("MQOPEN");
(*mqopen)(Hconn,&ObjDesc,Options,&Hobj,&CompCode,&Reason);
...
```

Figure 36. Dynamic linking using C language in the IMS environment

```
...
      DCL CSQBOPEN ENTRY EXT OPTIONS(ASSEMBLER INTER);
...
      FETCH CSQBOPEN;

      CALL CSQBOPEN(HQM,
                    MQOD,
                    OPTIONS,
                    HOBJ,
                    COMPCODE,
                    REASON);

      RELEASE CSQBOPEN;
```

Figure 37. Dynamic linking using PL/I in the batch environment

```
...
      DCL MQOPEN   ENTRY EXT OPTIONS(ASSEMBLER INTER);
...
      FETCH MQOPEN;

      CALL   MQOPEN(HQM,
                    MQOD,
                    OPTIONS,
                    HOBJ,
                    COMPCODE,
                    REASON);

      RELEASE   MQOPEN;
```

Figure 38. Dynamic linking using PL/I in the IMS environment

RRS Considerations

WebSphere MQ provides two different stubs for batch programs which need RRS coordination - see "RRS batch adapter" on page 271. The difference in behavior of subsequent API calls is determined at MQCONN time by the batch adapter from information passed by the stub routine on the MQCONN or MQCONNX API. This means that dynamic API calls are available for batch programs which need RRS coordination, provided that the initial connection to MQ has been done via the appropriate stub. The following example illustrates this:

```
       WORKING-STORAGE SECTION.
           05 WS-MQOPEN       PIC X(8) VALUE 'MQOPEN' .
.
.
.
       PROCEDURE DIVISION.
.
.
```

```
.
       *
       * Static call to MQCONN must be resolved by linkage edit to
       * CSQBRSTB or CSQBRRSI for RRS coordination
       *
           CALL 'MQCONN' USING W00-QMGR
                               W03-HCONN
                               W03-COMPCODE
                               W03-REASON.
.
.
.
       *
           CALL WS-MQOPEN  WS-HCONN
                           MQOD
                           WS-OPTIONS
                           WS-HOBJ
                           WS-COMPCODE
                           WS-REASON.
```

Debugging your programs

The main aids to debugging WebSphere MQ for z/OS application programs are the reason codes returned by each API call.

For a list of these, including ideas for corrective action, see:
- WebSphere MQ for z/OS Messages and Codes for WebSphere MQ for z/OS
- WebSphere MQ Messages for all other WebSphere MQ platforms

This chapter also suggests other debugging tools to use in particular environments.

Debugging CICS programs

You can use the CICS Execution Diagnostic Facility (CEDF) to test your CICS programs interactively without having to modify the program or program-preparation procedure.

For more information about EDF, see the *CICS Transaction Server for OS/390 CICS Application Programming Guide*.

CICS trace:

You will probably also find it helpful to use the CICS Trace Control transaction (CETR) to control CICS trace activity.

For more information about CETR, see *CICS Transaction Server for OS/390 CICS-Supplied Transactions* manual.

To determine whether CICS trace is active, display connection status using the CKQC panel. This panel also shows the trace number.

To interpret CICS trace entries, see Table 25 on page 383.

The CICS trace entry for these values is AP0*xxx* (where *xxx* is the trace number specified when the CICS adapter was enabled). All trace entries except CSQCTEST are issued by CSQCTRUE. CSQCTEST is issued by CSQCRST and CSQCDSP.

Table 25. CICS adapter trace entries

Name	Description	Trace sequence	Trace data
CSQCABNT	Abnormal termination	Before issuing END_THREAD ABNORMAL to WebSphere MQ. This is because of the end of the task and an implicit backout could be performed by the application. A ROLLBACK request is included in the END_THREAD call in this case.	Unit of work information. You can use this information when finding out about the status of work. (For example, it can be verified against the output produced by the DISPLAY THREAD command, or the WebSphere MQ for z/OS log print utility.)
CSQCBACK	Syncpoint backout	Before issuing BACKOUT to WebSphere MQ for z/OS. This is due to an explicit backout request from the application.	Unit of work information.
CSQCCCRC	Completion code and reason code	After unsuccessful return from API call.	Completion code and reason code.
CSQCCOMM	Syncpoint commit	Before issuing COMMIT to WebSphere MQ for z/OS. This can be due to a single-phase commit request or the second phase of a two-phase commit request. The request is due to a explicit syncpoint request from the application.	Unit of work information.
CSQCEXER	Execute resolve	Before issuing EXECUTE_RESOLVE to WebSphere MQ for z/OS.	The unit of work information of the unit of work issuing the EXECUTE_RESOLVE. This is the last indoubt unit of work in the resynchronization process.
CSQCGETW	GET wait	Before issuing CICS wait.	Address of the ECB to be waited on.
CSQCGMGD	GET message data	After successful return from MQGET.	Up to 40 bytes of the message data.
CSQCGMGH	GET message handle	Before issuing MQGET to WebSphere MQ for z/OS.	Object handle.
CSQCGMGI	Get message ID	After successful return from MQGET.	Message ID and correlation ID of the message.
CSQCINDL	Indoubt list	After successful return from the second INQUIRE_INDOUBT.	The indoubt units of work list.
CSQCINDO	IBM use only		
CSQCINDS	Indoubt list size	After successful return from the first INQUIRE_INDOUBT and the indoubt list is not empty.	Length of the list. Divided by 64 gives the number of indoubt units of work.
CSQCINQH	INQ handle	Before issuing MQINQ to WebSphere MQ for z/OS.	Object handle.
CSQCLOSH	CLOSE handle	Before issuing MQCLOSE to WebSphere MQ for z/OS.	Object handle.
CSQCLOST	Disposition lost	During the resynchronization process, CICS informs the adapter that it has been cold started so no disposition information regarding the unit of work being resynchronized is available.	Unit of work ID known to CICS for the unit of work being resynchronized.

Table 25. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCNIND	Disposition not indoubt	During the resynchronization process, CICS informs the adapter that the unit of work being resynchronized should not have been indoubt (that is, perhaps it is still running).	Unit of work ID known to CICS for the unit of work being resynchronized.
CSQCNORT	Normal termination	Before issuing END_THREAD NORMAL to WebSphere MQ for z/OS. This is due to the end of the task and therefore the application might perform an implicit syncpoint commit. A COMMIT request is included in the END_THREAD call in this case.	Unit of work information.
CSQCOPNH	OPEN handle	After successful return from MQOPEN.	Object handle.
CSQCOPNO	OPEN object	Before issuing MQOPEN to WebSphere MQ for z/OS.	Object name.
CSQCPMGD	PUT message data	Before issuing MQPUT to WebSphere MQ for z/OS.	Up to 40 bytes of the message data.
CSQCPMGH	PUT message handle	Before issuing MQPUT to WebSphere MQ for z/OS.	Object handle.
CSQCPMGI	PUT message ID	After successful MQPUT from WebSphere MQ for z/OS.	Message ID and correlation ID of the message.
CSQCPREP	Syncpoint prepare	Before issuing PREPARE to WebSphere MQ for z/OS in the first phase of two-phase commit processing. This call can also be issued from the distributed queuing component as an API call.	Unit of work information.
CSQCP1MD	PUTONE message data	Before issuing MQPUT1 to WebSphere MQ for z/OS.	Up to 40 bytes of data of the message.
CSQCP1MI	PUTONE message ID	After successful return from MQPUT1.	Message ID and correlation ID of the message.
CSQCP1ON	PUTONE object name	Before issuing MQPUT1 to WebSphere MQ for z/OS.	Object name.
CSQCRBAK	Resolved backout	Before issuing RESOLVE_ROLLBACK to WebSphere MQ for z/OS.	Unit of work information.
CSQCRCMT	Resolved commit	Before issuing RESOLVE_COMMIT to WebSphere MQ for z/OS.	Unit of work information.

Table 25. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCRMIR	RMI response	Before returning to the CICS RMI (resource manager interface) from a specific invocation.	Architected RMI response value. Its meaning depends of the type of the invocation. These values are documented in the *CICS Transaction Server for OS/390 Customization Guide*. To determine the type of invocation, look at previous trace entries produced by the CICS RMI component.
CSQCRSYN	Resynchronization	Before the resynchronization process starts for the task.	Unit of work ID known to CICS for the unit of work being resynchronized.
CSQCSETH	SET handle	Before issuing MQSET to WebSphere MQ for z/OS.	Object handle.
CSQCTASE	IBM use only		
CSQCTEST	Trace test	Used in EXEC CICS ENTER TRACE call to verify the trace number supplied by the user or the trace status of the connection.	No data.
CSQCDCFF	IBM use only		

Debugging TSO programs

The following interactive debugging tools are available for TSO programs:

- TEST tool
- VS COBOL II interactive debugging tool
- INSPECT interactive debugging tool for C and PL/I programs

Using lightweight directory access protocol services with WebSphere MQ for Windows

This chapter explains what a directory service is and the part played by a directory access protocol (DAP). It also explains how WebSphere MQ applications can use a lightweight directory access protocol (LDAP) directory using a sample program as a guide.

Note: The sample program is designed for someone who is already familiar with LDAP.

What is a directory service?

A directory is a repository of information about objects, which is organized in such a way that it is easy to find the information on a specific object.

A common example is a telephone directory, where information (address and telephone number) is stored about people and companies. Another example is an address book for an e-mail system, where e-mail addresses, and optionally other information such as telephone numbers, are stored for people.

On computer systems, directories can store information about computer resources, such as printers or shared disks. For example you could use a directory to find out

where the nearest color printer is located. In a WebSphere MQ application a directory can be used to provide the association between an application service (such as accounts-receivable processing) and the queue to be used for messages requiring that service (possibly identified through the queue name and its host queue manager name).

Directories are implemented as client-server systems, where the directory server holds all the information and answers requests from clients. The clients could be user-interface programs, which provide the information directly to the user, or application programs which need to locate resources to complete their work. A Directory Service comprises the directory server, administrative programs, and the client libraries and programs that are needed to configure, update, and read the directory.

What is LDAP?

A brief explanation of Lightweight Directory Access Protocol (LDAP).

Many directory services exist, such as Novell Directory Services, DCE Cell Directory Service, Banyan StreetTalk, Windows Directory Services, X.500, and the address book services associated with e-mail products. X.500 was proposed as a standard for global directory services by the International Standards Organization (ISO). It requires an OSI protocol stack for its communications, and largely because of this, its use has been restricted to large organizations and academic institutions. An X.500 directory server communicates with its clients using the Directory Access Protocol (DAP).

LDAP (Lightweight Directory Access Protocol) was created as a simplified version of DAP. It is easier to implement, omits some of the lesser-used features of DAP, and runs over TCP/IP. As a result of these changes it is rapidly being adopted as the directory access protocol for most purposes, replacing the multitude of proprietary protocols previously used. LDAP clients can still access an X.500 server through a gateway (X.500 still requires the OSI protocol stack), or increasingly X.500 implementations typically include native support for LDAP as well as DAP access.

LDAP directories can be distributed and can use replication to enable efficient access to their contents.

For a more complete description of LDAP, see *Understanding LDAP*, an IBM Redbooks publication.

Using LDAP with WebSphere MQ

In WebSphere MQ configurations, the information that defines message and transmission queues is stored locally. This means that in a WebSphere MQ network the various definitions are distributed, with no central directory of this information being available for browsing. Remote messaging between WebSphere MQ applications is commonly achieved through the use of local definitions of remote queues. The application first issues an MQOPEN call using the name specified in the local definition of the remote queue. To put the message on the remote queue, the application then issues MQPUT, specifying the handle returned from the MQOPEN call. The remote queue definition supplies the name of the destination

queue, the destination queue manager, and optionally, a transmission queue. In this technique the application has to know at run-time the name specified in the local queue definition.

A variation on the above avoids the use of local definitions of remote queues. The application can specify the full destination queue name, which includes the remote queue manager name as part of the MQOPEN. The application therefore has to know these two names at runtime. Again the local queue manager must be correctly configured with the local queue definition, and with a suitably named (or default) transmission queue and an associated channel that delivers to the target.

In the case where both the source and target queue managers are defined as being members of the same cluster, the transmission queue and channel aspects of the above two scenarios can be ignored. If the target transmission queue is a cluster queue, a local definition of a remote queue is also not required. However, similarly to the previous cases described, the application must still know the name of the destination queue.

A directory service can be used to remove this application dependency on queue names (or the combination of queue and queue manager names). The mapping between application criteria and WebSphere MQ object names can be held in a directory and updated dynamically, and independently of applications. At run time the WebSphere MQ application that wants to send a message first queries the directory using application-based criteria, for example where: service_name = “accounts receivable”, retrieves the relevant WebSphere MQ object names, and then uses these returned values in the MQOPEN call.

Another example of the use of a directory is for a company that has many small depots or offices, WebSphere MQ clients can be used to send messages to WebSphere MQ servers located in the larger offices. The clients need to know the name of the host machine, MQI channel, and queue name for each server that they send messages to. Occasionally it might be necessary to move a WebSphere MQ server to another machine; every client that communicates with the server would need to know about the change. An LDAP directory service could be used to store the names of the host machines (and the channel and queue names) and the client programs could retrieve the information from the directory whenever they want to send a message to a server. In this case only the directory needs to be updated if a host name (or channel or queue name) changed.

Multiple destinations for an application message could be stored in a directory, with the one chosen being dependent on availability or load-sharing considerations.

WebSphere MQ can also use an LDAP directory to store authentication information for use with Secure Sockets Layer (SSL). WebSphere MQ classes for Java can also store information in an LDAP directory.

LDAP sample program

The sample program is designed for someone who is familiar with LDAP and probably already uses it. It is intended to show how WebSphere MQ applications can use an LDAP directory.

Building the sample program

This program has been built and tested only on Windows using TCP/IP. As well as the general considerations mentioned in "Preparing C programs" on page 366, note the following points:

- This program is designed to run as a client program, so it should be linked with the MQIC.LIB library.
- As well as the WebSphere MQ header files and libraries, this program must be built using LDAP client header files and libraries. These are available from several locations, including the IBM eNetwork Web site at:

  `http://www.software.ibm.com/enetwork`

  For example, using the IBM eNetwork client, link the program with the LIBLDAPSTATICE.LIB and LIBLBERSTATICSSL.LIB libraries.

Configuring the directory

Before the sample program can be run, an LDAP Directory Server must be configured with sample data.

The file MQuser.ldif, in the `tools\c\samples` directory, contains some sample data in LDIF (LDAP Data Interchange Format). You can edit this file to suit your needs. It contains data for a fictitious company called MQuser that has a Transport Department comprising three offices. Each of these offices has a machine that runs a WebSphere MQ server.

As a minimum, you must edit the three lines that contain the host names of the machines running the WebSphere MQ servers: lines 18, 27, and 36:

```
host: LondonHost
  ...
host: SydneyHost
  ...
host: WashingtonHost
```

You must change `LondonHost`, `SydneyHost`, and `WashingtonHost` to the names of three of your machines that run WebSphere MQ servers. You can also change the channel and queue names if you want (the sample uses names of the system defaults). You might also want to increase or decrease the number of offices in the sample data.

Configuring the IBM eNetwork LDAP server

Refer to the eNetwork LDAP Directory Administrator's Guide for information about installing the directory. In the chapter "Installing and Configuring Server", work through the sections "Installing Server" and "Basic Server Configuration". If necessary, read through the chapter "Administrator Interface" to familiarize yourself with how the interface works.

In the chapter "Configuring - How Do I", follow the instructions for starting up the administrator, then work through the section "Configure Database" and create a default database. Skip the section "Configure replica" and using the section "Work with Suffixes", add a suffix "o=MQuser".

Before adding any entries to the database, you must extend the directory schema by adding some attribute definitions and an objectclass definition. This is described in the eNetwork LDAP Directory Administrator's Guide in the chapter "Reference Information" under the section "Directory Schema". Two sample files are included

to help you with this. The file mq.at.conf includes the attribute definitions that you must add to the file /etc/slapd.at.conf. Do this by including the sample file by editing slapd.at.conf and adding a line:

```
include <pathname>/mq.at.conf
```

Alternatively you can edit the file slapd.at.conf and add the contents of the sample file directly to it, that is, add the lines:

```
# MQ attribute definitions
attribute mqChannel           ces   mqChannel        1000   normal
attribute mqQueueManager      ces   mqQueueManager   1000   normal
attribute mqQueue             ces   mqQueue          1000   normal
attribute mqPort              cis   mqPort           64     normal
```

Similarly for the objectclass definition, you can either include the sample file by editing etc/slapd.oc.conf and add the line:

```
include <pathname>/mq.oc.conf
```

or you can add the contents of the sample file directly to slapd.oc.conf, that is, add the lines:

```
# MQ object classdefinition
objectclass mqApplication
      requires
              objectClass,
              cn,
              host,
              mqChannel,
              mqQueue
      allows
              mqQueueManager,
              mqPort,
              description,
              l,
              ou,
              seeAlso
```

You can now start the directory server (Administration, Server, Startup) and add the sample entries to it. To add the sample entries, go to the Administration, Add Entries page of the administrator, type in the full pathname of the sample file MQuser.ldif and click Submit.

The directory server is now running and loaded with data suitable for running the sample program.

Configuring the Netscape directory server

Using the Netscape Server Administration page, click on **Create New Netscape Directory Server**.

You should now be presented with a form containing configuration information. Change the Directory Suffix to **o=MQuser** and add a password for the Unrestricted User. You can also change any other information to suit your installation. Click **OK**, and the directory should be created successfully. Click **Return to Server Administration** and start the directory server. Click the directory name to start the Directory Server Administration server for the new directory.

Before adding any entries to the database, extend the directory schema by adding some attribute definitions and an objectclass definition. Click the **Schema** tab of the

Directory Server page. You are now presented with a form that allows you to add new attributes. Add the following attributes (leave the Attribute OID blank for all of them):

```
Attribute Name              Syntax
--------------              ------
mqChannel                   Case Exact String
mqQueueManager              Case Exact String
mqQueue                     Case Exact String
mqPort                      Integer
```

Add a new objectClass by clicking **Create ObjectClass** in the side panel. Enter **mqApplication** as the ObjectClass Name, select **applicationProcess** as the parent ObjectClass and leave the **ObjectClass OID** blank. Now add some attributes to the objectClass. Select **host**, **mqChannel**, and **mqQueue** as Required Attributes, and select **mqQueueManager** and **mqPort** as Allowed attributes. Press the **Create New ObjectClass** button to create the objectClass.

To add the sample data, click the **Database Management** tab and select **Add Entries** from the side panel. Enter the pathname of the sample data file <pathname>\MQuser.ldif, enter the password, and click on **OK**.

The sample program runs as an unauthorized user, and by default the Netscape Directory does not allow unauthorized users to search the directory. Change this by clicking the **Access Control** tab. Enter the password for the Unrestricted User and click **OK** to load in the access control entries for the directory. These should currently be empty. Press the **New ACI** button to create a new access control entry. In the entry box that appears, click **Deny** (which is underlined) and in the resultant dialog box, change it to **Allow**. Add a name, for example, **MQuser-access**, and click **choose a suffix** to select **o=MQuser**. Enter **o=MQuser** as the target, enter the password for the Unrestricted User, and click on the **Submit** button.

The directory server is now running and loaded with data suitable for running the sample program.

Running the sample program

You should now have an LDAP Directory Server running and populated with the sample data. The data specifies three host machines, all of which should be running WebSphere MQ servers. Ensure that the default queue manager is running on each machine (unless you changed the sample data to specify a different queue manager).

Also, start the WebSphere MQ listener program on each machine; the sample uses TCP/IP with the default WebSphere MQ port number, so you can start the listener with the command:

```
runmqlsr -t tcp
```

To test the sample, you might also want to run a program to read the messages arriving at each WebSphere MQ server, for example you could use the amqstrg sample program:

```
amqstrg SYSTEM.DEFAULT.LOCAL.QUEUE
```

The sample program uses three environment variables, one required and two optional. The required variable is LDAP_BASEDN, which specifies the base

Distinguished Name for the directory search. To work with the sample data, set this to `ou=Transport, o=MQuser`, for example, at a command prompt on Windows systems type:

```
set LDAP_BASEDN=ou=Transport, o=MQuser
```

The optional variables are LDAP_HOST and LDAP_VERSION. The LDAP_HOST variable specifies the name of the host where the LDAP server is running; it defaults to the local host if it is not specified. The LDAP_VERSION variable specifies the version of the LDAP protocol to be used, and can be either 2 or 3. Most LDAP servers now support version 3 of the protocol; they all support the older version 2. This sample works equally well with either version of the protocol, and if it is not specified it defaults to version 2.

You can now run the sample by typing the program name followed by the name of the WebSphere MQ application that you want to send messages to, in the case of the sample data the application names are London, Sydney, and Washington. For example, to send messages to the London application:

```
amqsldpc London
```

If the program fails to connect to the WebSphere MQ server, an appropriate error message appears. If it connects successfully you can start typing messages, each line that you type (terminated by <return> or <enter>) is sent as a separate message, an empty line ends the program.

Program design

The program has two distinct parts: the first part uses the environment variables and command line value to query an LDAP directory server; the second part establishes the WebSphere MQ connection using the information returned from the directory and sends the messages.

The LDAP calls used in the first part of the program differ slightly depending on whether LDAP version 2 or 3 is being used, and they are described in detail by the documentation that comes with the LDAP client libraries. This section gives a brief description.

The first part of the program checks that it has been called correctly and reads the environment variables. It then establishes a connection with the LDAP directory server at the specified host:

```
if (ldapVersion == LDAP_VERSION3)
{
  if ((ld = ldap_init(ldapHost, LDAP_PORT)) == NULL)
     ...
}
else
{
  if ((ld = ldap_open(ldapHost, LDAP_PORT)) == NULL )
     ...
}
```

When a connection has been established, the program sets some options on the server with the “ldap_set_option” call, and then authenticates itself to the server by binding to it:

```
if (ldapVersion == LDAP_VERSION3)
{
  if (ldap_simple_bind_s(ld, bindDN, password) != LDAP_SUCCESS)
     ...
}
else
```

```
{
  if (ldap_bind_s(ld, bindDN, password, LDAP_AUTH_SIMPLE) !=
      LDAP_SUCCESS)
     ...
}
```

In the sample program `bindDN` and `password` are set to NULL, which means that the program authenticates itself as an anonymous user, that is, it does not have any special access rights and can access only information that is publicly available. In practice, most organizations restrict access to the information that they store in directories so that only authorized users can access it.

The first parameter to the bind call `ld` is a handle that is used to identify this particular LDAP session throughout the rest of the program. After authenticating, the program searches the directory for entries that match the application name:

```
rc = ldap_search_s(ld,                /* LDAP Handle             */
                   baseDN,            /* base distinguished name */
                   LDAP_SCOPE_ONELEVEL, /* one-level search       */
                   filterPattern,     /* filter search pattern   */
                   attrs,             /* attributes required     */
                   FALSE,             /* NOT attributes only     */
                   &ldapResult);      /* search result           */
```

This is a simple synchronous call to the server that returns the results directly. There are other types of search that are more appropriate for complex queries or when a large number of results is expected. The first parameter to the search is the handle `ld` that identifies the session. The second parameter is the base distinguished name, which specifies where in the directory the search is to begin, and the third parameter is the scope of the search, that is, which entries relative to the starting point are searched. These two parameters together define which entries in the directory are searched. The next parameter, `filterPattern` specifies what we are searching for. The `attrs` parameter lists the attributes that we want to get back from the object when we have found it. The next attribute says whether we want just the attributes or their values as well; setting this to FALSE means that we want the attribute values. The final parameter is used to return the result.

The result could contain many directory entries, each with the specified attributes and their values. We have to extract the values that we want from the result. In this sample program we only expect one entry to be found, so we only look at the first entry in the result:

```
ldapEntry = ldap_first_entry(ld, ldapResult);
```

This call returns a handle that represents the first entry, and we set up a for loop to extract all the attributes from the entry:

```
for (attribute = ldap_first_attribute(ld, ldapEntry, &ber);
     attribute != NULL;
     attribute = ldap_next_attribute(ld, ldapEntry, ber ))
{
```

For each of these attributes, we extract the values associated with it. Again we only expect one value per attribute, so we only use the first value; we determine which attribute we have and store the value in the appropriate program variable:

```
values = ldap_get_values(ld, ldapEntry, attribute);
if (values != NULL && values[0] != NULL)
{
  if (stricmp(attribute, MQ_HOST_ATTR) == 0)
  {
    mqHost = strdup(values[0]);
    ...
```

Finally we tidy up by freeing memory (ldap_value_free, ldap_memfree, ldap_msgfree) and close the session by *unbinding* from the server:

```
ldap_unbind(ld);
```

We check that we have found all the WebSphere MQ values that we need from the directory, and if so we call sendMessages() to connect to the WebSphere MQ server and send the WebSphere MQ messages.

The second part of the sample program is the sendMessages() routine that contains all the WebSphere MQ calls. This is modelled on the amqsput0 sample program, the differences being that the parameters to the program have been extended and MQCONNX is used instead of the MQCONN call.

Chapter 4. Sample WebSphere MQ programs

Sample programs (all platforms except z/OS)

This chapter describes the sample programs delivered with WebSphere MQ, written in C and COBOL. The samples demonstrate typical uses of the Message Queue Interface (MQI).

The samples are not intended to demonstrate general programming techniques, so some error checking that you might want to include in a production program has been omitted. However, these samples are suitable for use as a base for your own message queuing programs.

The source code for all the samples is provided with the product; this source includes comments that explain the message queuing techniques demonstrated in the programs.

C++ sample programs: See WebSphere MQ Using C++ for a description of the sample programs available in C++.

RPG sample programs: See the WebSphere MQ for i5/OS Application Programming Reference (ILE/RPG) for a description of the sample programs available in RPG.

The names of the samples start with the prefix amq. The fourth character indicates the programming language, and the compiler where necessary.

s	C language
0	COBOL language on both IBM and Micro Focus compilers
i	COBOL language on IBM compilers only
m	COBOL language on Micro Focus compilers only

This chapter introduces the sample programs, under these headings:

- "Running the samples using remote queues" on page 430
- "Database coordination samples" on page 430
- "The CICS transaction sample" on page 437
- "TUXEDO samples" on page 437
- "Encina sample program" on page 449
- "Dead-letter queue handler sample" on page 449
- "The Connect sample program" on page 450
- "The API exit sample program" on page 451
- "Using the SSPI security exit on Windows systems" on page 452

Features demonstrated in the sample programs

The following tables show the techniques demonstrated by the WebSphere MQ sample programs on all systems except z/OS (see "Sample programs for WebSphere MQ for z/OS" on page 452). All the samples open and close queues using the MQOPEN and MQCLOSE calls, so these techniques are not listed separately in the tables. See the heading that includes the platform that you are interested in:

- "Samples for UNIX systems"
- "Samples for WebSphere MQ for Windows" on page 398
- "Visual Basic samples for WebSphere MQ for Windows" on page 399
- "Samples for WebSphere MQ for i5/OS" on page 400

Samples for UNIX systems

Table 26 shows the techniques demonstrated by the sample programs for WebSphere MQ on UNIX systems.

Table 26. WebSphere MQ on UNIX sample programs demonstrating use of the MQI (C and COBOL)

Technique	C (source) (1 on page 398)	COBOL (source) (2 on page 398)	C (executable)	Client (executable) (3 on page 398)
Putting messages using the MQPUT call	amqsput0	amq0put0	amqsput	amqsputc
Putting a single message using the MQPUT1 call	amqsinqa amqsecha	amqminqx amqmechx amqiinqx amqiechx amqvinqx amqviechx	amqsinq amqsech	amqsechc
Putting messages to a distribution list (4 on page 398)	amqsptl0	amq0ptl0.cbl	amqsptl	amqsptlc
Replying to a request message	amqsinqa	amqminqx amqiinqx amqvinqx	amqsinq	no sample
Getting messages (no wait)	amqsgbr0	amq0gbr0	amqsgbr	no sample
Getting messages (wait with a time limit)	amqsget0	amq0get0	amqsget	amqsgetc
Getting messages (unlimited wait)	amqstrg0	no sample	amqstrg	amqstrgc
Getting messages (with data conversion)	amqsecha	no sample	amqsech	no sample
Putting Reference Messages to a queue (4 on page 398)	amqsprma	no sample	amqsprm	amqsprmc

Table 26. WebSphere MQ on UNIX sample programs demonstrating use of the MQI (C and COBOL) (continued)

Technique	C (source) (1 on page 398)	COBOL (source) (2 on page 398)	C (executable)	Client (executable) (3 on page 398)
Getting Reference Messages from a queue (4 on page 398)	amqsgrma	no sample	amqsgrm	amqsgrmc
Reference Message channel exit (4 on page 398)	amqsqrma amqsxrma	no sample	amqsxrm	no sample
Browsing first 20 characters of a message	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Browsing complete messages	amqsbcg0	no sample	amqsbcg	amqsbcgc
Using a shared input queue	amqsinqa	amqminqx amqiinqx amqvinqx	amqsinq	amqsinqc
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc
Using the MQINQ call	amqsinqa	amqminqx amqiinqx amqvinqx	amqsinq	no sample
Using the MQSET call	amqsseta	amqmsetx amqisetx amqvsetx	amqsset	amqssetc
Using a reply-to queue	amqsreq0	amq0req0	amqsreq	amqsreqc
Requesting message exceptions	amqsreq0	amq0req0	amqsreq	no sample
Accepting a truncated message	amqsgbr0	amq0gbr0	amqsgbr	no sample
Using a resolved queue name	amqsgbr0	amq0gbr0	amqsgbr	no sample
Triggering a process	amqstrg0	no sample	amqstrg	amqstrgc
Using data conversion	(5 on page 398)	no sample	no sample	no sample
WebSphere MQ (coordinating XA-compliant database managers) accessing a single database using SQL	amqsxas0.sqc DB2 amqsxas0.ec Informix	amq0xas0.sqb	no sample	no sample
WebSphere MQ (coordinating XA-compliant database managers) accessing two databases using SQL	amqsxag0.c amqsxab0.sqc amqsxaf0.sqc	amq0xag0.cbl amq0xab0.sqb amq0xaf0.sqb	no sample	no sample
CICS transaction (6 on page 398)	amqscic0.ccs	no sample	amqscic0	no sample
Encina transaction (4 on page 398)	amqsxae0	no sample	amqsxae0	no sample
TUXEDO transaction to put messages (7 on page 398)	amqstxpx	no sample	no sample	no sample
TUXEDO transaction to get messages (7 on page 398)	amqstxgx	no sample	no sample	no sample
Server for TUXEDO (7 on page 398)	amqstxsx	no sample	no sample	no sample
Dead-letter queue handler	(8 on page 398)	no sample	amqsdlq	no sample
From an MQI client, putting a message	no sample	no sample	no sample	amqsputc
From an MQI client, getting a message	no sample	no sample	no sample	amqsgetc
Connecting to the queue manager using MQCONNX	amqscnxc	no sample	no sample	amqscnxc
Using API exits	amqsaxe0.c	no sample	amqsaxe	no sample
Cluster workload balancing exit	amqswlm0.c	no sample	amqswlm	no sample

Table 26. WebSphere MQ on UNIX sample programs demonstrating use of the MQI (C and COBOL) (continued)

Technique	C (source) (1)	COBOL (source) (2)	C (executable)	Client (executable) (3)
Putting messages asynchronously and getting status using the MQSTAT call	amqsapt0.c	no sample	amqsapt	amqsaptc

Notes:

1. The executable version of the WebSphere MQ client samples share the same source as the samples that run in a server environment.
2. COBOL is not supported by WebSphere MQ for Linux. Compile programs beginning 'amqm' with the Micro Focus COBOL compiler, those beginning 'amqi' with the IBM COBOL compiler, and those beginning 'amq0' with either.
3. The executable versions of the WebSphere MQ client samples are not available on WebSphere MQ for HP-UX.
4. Supported on WebSphere MQ for AIX, WebSphere MQ for HP-UX, and WebSphere MQ for Solaris only.
5. On WebSphere MQ for AIX, WebSphere MQ for HP-UX, and WebSphere MQ for Solaris this program is called amqsvfc0.c.
6. CICS is supported by WebSphere MQ for AIX and WebSphere MQ for HP-UX only.
7. TUXEDO is not supported by WebSphere MQ for Linux
8. The source for the dead-letter queue handler is made up of several files and provided in a separate directory.

Samples for WebSphere MQ for Windows

Table 27 shows the techniques demonstrated by the sample programs for WebSphere MQ for Windows.

Table 27. WebSphere MQ for Windows sample programs demonstrating use of the MQI (C and COBOL)

Technique	C (source)	COBOL (source)	C (executable)	Client (executable)
Putting messages using the MQPUT call	amqsput0	amq0put0	amqsput	amqsputc
Putting a single message using the MQPUT1 call	amqsinqa amqsecha	amqminq2 amqmech2 amqiinq2 amqiech2	amqsinq amqsech	amqsinqc amqsechc
Putting messages to a distribution list	amqsptl0	amq0ptl0.cbl	amqsptl	amqsptlc
Replying to a request message	amqsinqa	amqminq2 amqiinq2	amqsinq	amqsinqc
Getting messages (no wait)	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Getting messages (wait with a time limit)	amqsget0	amq0get0	amqsget	amqsgetc
Getting messages (unlimited wait)	amqstrg0	no sample	amqstrg	amqstrgc
Getting messages (with data conversion)	amqsecha	no sample	amqsech	amqsechc
Putting Reference Messages to a queue	amqsprma	no sample	amqsprm	amqsprmc
Getting Reference Messages from a queue	amqsgrma	no sample	amqsgrm	amqsgrmc
Reference Message channel exit	amqsqrma amqsxrma	no sample	amqsxrm	no sample
Browsing first 20 characters of a message	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Browsing complete messages	amqsbcg0	no sample	amqsbcg	amqsbcgc
Using a shared input queue	amqsinqa	amqminq2 amqiinq2	amqsinq	amqsinqc
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc

Table 27. WebSphere MQ for Windows sample programs demonstrating use of the MQI (C and COBOL) (continued)

Technique	C (source)	COBOL (source)	C (executable)	Client (executable)
Using the MQINQ call	amqsinqa	amqminq2 amqiinq2	amqsinq	amqsinqc
Using the MQSET call	amqsseta	amqmset2 amqiset2	amqsset	amqssetc
Using a reply-to queue	amqsreq0	amq0req0	amqsreq	amqsreqc
Requesting message exceptions	amqsreq0	amq0req0	amqsreq	amqsreqc
Accepting a truncated message	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Using a resolved queue name	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Triggering a process	amqstrg0	no sample	amqstrg	amqstrgc
Using data conversion	amqsvfc0	no sample	no sample	no sample
WebSphere MQ (coordinating XA-compliant database managers) accessing a single database using SQL	amqsxas0.sqc DB2 amqsxas0.ec Informix	amq0xas0.sqb	no sample	no sample
WebSphere MQ (coordinating XA-compliant database managers) accessing two databases using SQL	amqsxag0.c amqsxab0.sqc amqsxaf0.sqc	amq0xag0.cbl amq0xab0.sqb amq0xaf0.sqb	no sample	no sample
TUXEDO transaction to put messages	amqstxpx	no sample	no sample	no sample
TUXEDO transaction to get messages	amqstxgx	no sample	no sample	no sample
Server for TUXEDO	amqstxsx	no sample	no sample	no sample
Dead-letter queue handler	(2)	no sample	amqsdlq	no sample
From a WebSphere MQ client, putting a message	no sample	no sample	no sample	amqsputc
From a WebSphere MQ client, getting a message	no sample	no sample	no sample	amqsgetc
Connecting to the queue manager using MQCONNX	amqscnxc	no sample	no sample	amqscnxc
Using API exits	amqsaxe0	no sample	amqsaxe	no sample
Cluster workload balancing	amqswlm0	no sample	amqswlm	no sample
SSPI security routines	amqsspin	no sample	amqrspin.dll	amqrspin.dll
Putting messages asynchronously and getting status using the MQSTAT call	amqsapt0	no sample	amqsapt	amqsaptc

Notes:

1. The executable version on Windows systems is for TXSeries for Windows V5.1.
2. The source for the dead-letter queue handler is made up of several files and provided in a separate directory.

Visual Basic samples for WebSphere MQ for Windows

Table 28 on page 400 shows the techniques demonstrated by the WebSphere MQ for Windows sample programs.

A project can contain several files. When you open a project within Visual Basic, the other files are loaded automatically. No executable programs are provided.

All the sample projects, except mqtrivc.vbp, are set up to work with the WebSphere MQ server. To find out how to change the sample projects to work with the WebSphere MQ clients see "Preparing Visual Basic programs" on page 370.

Table 28. WebSphere MQ for Windows sample programs demonstrating use of the MQI (Visual Basic)

Technique	Project file name
Putting messages using the MQPUT call	amqsputb.vbp
Getting messages using the MQGET call	amqsgetb.vbp
Browsing a queue using the MQGET call	amqsbcgb.vbp
Simple MQGET and MQPUT sample (client)	mqtrivc.vbp
Simple MQGET and MQPUT sample (server)	mqtrivs.vbp
Putting and getting strings and user-defined structures using MQPUT and MQGET	strings.vbp
Using PCF structures to start and stop a channel	pcfsamp.vbp
Creating a queue using the MQAI	amqsaicq.vbp
Listing a queue manager's queues using the MQAI	amqsailq.vbp
Monitoring events using the MQAI	amqsaiem.vbp

Samples for WebSphere MQ for i5/OS

Table 29 shows the techniques demonstrated by the WebSphere MQ for i5/OS sample programs. Some techniques occur in more than one sample program, but only one program is listed in the table.

Table 29. WebSphere MQ for i5/OS sample programs demonstrating use of the MQI (C and COBOL)

Technique	C (source) (1 on page 401)	COBOL (source) (2 on page 401)	RPG (source) (3 on page 401)
Putting messages using the MQPUT call	AMQSPUT0	AMQ0PUT4	AMQ3PUT4
Putting messages from a data file using the MQPUT call	AMQSPUT4	no sample	no sample
Putting a single message using the MQPUT1 call	AMQSINQ4, AMQSECH4	AMQ0INQ4, AMQ0ECH4	AMQ3INQ4, AMQ3ECH4
Putting messages to a distribution list	AMQSPTL4	no sample	no sample
Replying to a request message	AMQSINQ4	AMQ0INQ4	AMQ3INQ4
Getting messages (no wait)	AMQSGBR4	AMQ0GBR4	AMQ3GBR4
Getting messages (wait with a time limit)	AMQSGET4	AMQ0GET4	AMQ3GET4
Getting messages (unlimited wait)	AMQSTRG4	no sample	AMQ3TRG4
Getting messages (with data conversion)	AMQSECH4	AMQ0ECH4	AMQ3ECH4
Putting Reference Messages to a queue	AMQSPRM4	no sample	no sample
Getting Reference Messages from a queue	AMQSGRM4	no sample	no sample
Reference Message channel exit	AMQSQRM4, AMQSXRM4	no sample	no sample
Message exit	AMQSCMX4	no sample	no sample
Browsing first 49 characters of a message	AMQSGBR4	AMQ0GBR4	AMQ3GBR4
Browsing complete messages	AMQSBCG4	no sample	no sample
Using a shared input queue	AMQSINQ4	AMQ0INQ4	AMQ3INQ4

Table 29. WebSphere MQ for i5/OS sample programs demonstrating use of the MQI (C and COBOL) (continued)

Technique	C (source) (1)	COBOL (source) (2)	RPG (source) (3)
Using an exclusive input queue	AMQSREQ4	AMQ0REQ4	AMQ3REQ4
Using the MQINQ call	AMQSINQ4	AMQ0INQ4	AMQ3INQ4
Using the MQSET call	AMQSSET4	AMQ0SET4	AMQ3SET4
Using a reply-to queue	AMQSREQ4	AMQ0REQ4	AMQ3REQ4
Requesting message exceptions	AMQSREQ4	AMQ0REQ4	AMQ3REQ4
Accepting a truncated message	AMQSGBR4	AMQ0GBR4	AMQ3GBR4
Using a resolved queue name	AMQSGBR4	AMQ0GBR4	AMQ3GBR4
Triggering a process	AMQSTRG4	no sample	AMQ3TRG4
Trigger server	AMQSERV4	no sample	AMQ3SRV4
Using a trigger server (including CICS transactions)	AMQSERV4	no sample	AMQ3SRV4
Using data conversion	AMQSVFC4	no sample	no sample
Using API exits	AMQSAXE0	no sample	no sample
Cluster workload balancing	AMQSWLM0	no sample	no sample
Putting messages asynchronously and getting status using the MQSTAT call	AMQSAPT0	no sample	no sample

Notes:

1. Source for the C samples is in the file QMQMSAMP/QCSRC. Include files exist as members in the file QMQM/H.
2. Source for the COBOL samples are in the files QMQMSAMP/QCBLLESRC. The members are named AMQ0xxx4, where xxx indicates the sample function.
3. Source for the RPG samples is in QMQMSAMP/QRPGLESRC. Members are named AMQ3xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name has the suffix G.

In addition to these, the WebSphere MQ for i5/OS sample option includes a sample data file, which you use as input to the sample programs, AMQSDATA and sample CL programs that demonstrate administration tasks. The CL samples are described in the WebSphere MQ for i5/OS System Administration Guide. You could use the sample CL program `amqsamp4` to create queues to use with the sample programs described in this chapter.

Preparing and running the sample programs

The following sections help you to find the samples that you need to run on the different platforms.

i5/OS

The source for WebSphere MQ for i5/OS sample programs are provided in library QMQMSAMP as members of QCSRC, QCLSRC, QCBLLESRC, and QRPGLESRC.

To run the samples use either the C executable versions, supplied in the library QMQM, or compile them as you would any other WebSphere MQ application. For more information see “Running the sample programs” on page 403.

UNIX systems

Table 30. Where to find the samples for WebSphere MQ on UNIX systems

Content	Directory
source files	/**mqmtop**/samp
dead-letter queue handler source files	/**mqmtop**/samp/dlq
executable files	/**mqmtop**/samp/bin
Note: For WebSphere MQ for AIX **mqmtop** is /usr/mqm; for WebSphere MQ for other UNIX systems **mqmtop** is /opt/mqm.	

The WebSphere MQ on UNIX systems sample files are in the directories listed in Table 30 if the defaults were used at installation time. To run the samples, either use the executable versions supplied or compile the source versions as you would any other applications, using an ANSI compiler. For information on how to do this, see "Running the sample programs" on page 403.

Windows systems

Table 31. Where to find the samples for WebSphere MQ for Windows

Content	Directory
C source code	<install_location>\Tools\C\Samples
Source code for dead-letter handler sample	<install_location>\Tools\C\Samples\DLQ
COBOL source code	<install_location>\Tools\Cobol\Samples
C executable files[1]	<install_location>\ Tools\C\Samples\Bin (32-bit versions) <install_location>\ Tools\C\Samples\Bin64 (64-bit versions)
Sample MQSC files	<install_location>\Tools\MQSC\Samples
Visual Basic source code	<install_location>\Tools\VB\SampVB6
.NET samples	<install_location>\Tools\dotnet\Samples

Note:

1. 64-bit versions are available of some C executable file samples.

The WebSphere MQ for Windows sample files are in the directories listed in Table 31 if the defaults were used at installation time; the installation drive defaults to <c:>. To run the samples, either use the executable versions supplied or compile the source versions as you would any other WebSphere MQ for Windows applications. For information on how to do this, see "Running the sample programs" on page 403.

Running the sample programs

Before you can run any of the sample programs, create a queue manager and set up the default definitions. This is explained in WebSphere MQ System Administration Guide.

On all platforms except i5/OS:

The samples need a set of queues to work with. Either use your own queues or run the sample MQSC file amqscos0.tst to create a set.

To do this on UNIX systems, enter:

- `runmqsc QManagerName <amqscos0.tst >/tmp/sampobj.out`

Check the `sampobj.out` file to ensure that there are no errors.

To do this on Windows systems enter:

- `runmqsc QManagerName  <amqscos0.tst > sampobj.out`

Check the `sampobj.out` file to ensure that there are no errors. This file is in your current directory.

You can now run the sample applications. Enter the name of the sample application followed by any parameters, for example:

- `amqsput myqueue qmanagername`

where `myqueue` is the name of the queue on which the messages are going to be put, and `qmanagername` is the queue manager that owns `myqueue`.

See the description of the individual samples for information on the parameters that each of them expects.

On i5/OS:

You can use your own queues when you run the samples, or you can run the sample program AMQSAMP4 to create some sample queues. The source for this program is shipped in file QCLSRC in library QMQMSAMP. It can be compiled using the CRTCLPGM command.

To call one of the sample programs using data from member PUT in file AMQSDATA of library QMQMSAMP, use a command like:

```
CALL PGM(QMQM/AMQSPUT4) PARM('QMQMSAMP/AMQSDATA(PUT)')
```

The sample data only applies to the C/400® sample programs.

Note: For a compiled module to use the IFS file system, specify the option SYSIFCOPT(*IFSIO) on CRTCMOD, then the file name, passed as a parameter, must be specified in the following format:

```
home/me/myfile
```

Length of queue name:

For the COBOL sample programs, when you pass queue names as parameters, you must provide 48 characters, padding with blank characters if necessary. Anything other than 48 characters causes the program to fail with reason code 2085.

Inquire, Set, and Echo examples:

For the Inquire, Set, and Echo examples, the sample definitions trigger the C versions of these samples.

If you want the COBOL versions you must change the process definitions:

- SYSTEM.SAMPLE.INQPROCESS
- SYSTEM.SAMPLE.SETPROCESS
- SYSTEM.SAMPLE.ECHOPROCESS

On Windows systems and UNIX do this by editing the amqscos0.tst file and changing the C executable file names to the COBOL executable file names before using the **runmqsc** command above.

On i5/OS, you can use the CHGMQMPRC command (described in the WebSphere MQ for i5/OS System Administration Guide), or edit and run AMQSAMP4 with the alternative definition.

The Put sample programs

The Put sample programs put messages on a queue using the MQPUT call.

See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

Running the amqsput and amqsputc samples

These programs each take 2 parameters:

1. The name of the target queue (required)
2. The name of the queue manager (optional)

If a queue manager is not specified, amqsput connects to the default queue manager and amqsputc connects to the queue manager identified by an environment variable or the client channel definition file. To run these programs, enter one of the following:

- `amqsput myqueue qmanagername`
- `amqsputc myqueue qmanagername`

where `myqueue` is the name of the queue on which the messages are going to be put, and `qmanagername` is the queue manager that owns `myqueue`.

Running the amq0put sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target queue
```

It takes input from StdIn and adds each line of input to the target queue. A blank line indicates there is no more data.

Running the AMQSPUT4 C sample

The C program creates messages by reading data from a member of a source file.

You must specify the name of the file as a parameter when you start the program. The structure of the file must be:

```
queue name
text of message 1
text of message 2
```

```
⋮
   text of message n
   blank line
```

A sample of input for the put samples is supplied in library QMQMSAMP file AMQSDATA member PUT.

Note: Remember that queue names are case sensitive. All the queues created by the sample file create program AMQSAMP4 have names created in uppercase characters.

The C program puts messages on the queue named in the first line of the file; you can use the supplied queue SYSTEM.SAMPLE.LOCAL. The program puts the text of each of the following lines of the file into separate datagram messages, and stops when it reads a blank line at the end of the file.

Using the example data file the command is:

```
CALL PGM(QMQM/AMQSPUT4) PARM('QMQMSAMP/AMQSDATA(PUT)')
```

Running the AMQ0PUT4 COBOL sample

The COBOL program creates messages by accepting data from the keyboard.

To start the program, call the program and give the name of your target queue as a program parameter. The program accepts input from the keyboard into a buffer and creates a datagram message for each line of text. The program stops when you enter a blank line at the keyboard.

Design of the Put sample program

The program uses the MQOPEN call with the MQOO_OUTPUT option to open the target queue for putting messages.

If it cannot open the queue, the program outputs an error message containing the reason code returned by the MQOPEN call. To keep the program simple, on this and on subsequent MQI calls, the program uses default values for many of the options.

For each line of input, the program reads the text into a buffer and uses the MQPUT call to create a datagram message containing the text of that line. The program continues until either it reaches the end of the input or the MQPUT call fails. If the program reaches the end of the input, it closes the queue using the MQCLOSE call.

The Distribution List sample program

The Distribution List sample amqsptl0 gives an example of putting a message on several message queues. It is based on the MQPUT sample, amqsput0.

Running the Distribution List sample, amqsptl0

The Distribution List sample runs in a similar way to the Put samples.

It takes the following parameters:

- The names of the queues
- The names of the queue managers

These values are entered as pairs. For example:

```
amqsptl0 queue1 qmanagername1 queue2 qmanagername2
```

The queues are opened using MQOPEN and messages are put to the queues using MQPUT. Reason codes are returned if any of the queue or queue manager names are not recognized.

Remember to define channels between queue managers so that messages can flow between them. The sample program does not do that for you.

Design of the Distribution List sample

Put Message Records (MQPMRs) specify message attributes for each destination. The sample provides values for *MsgId* and *CorrelId*, and these override the values specified in the MQMD structure.

The *PutMsgRecFields* field in the MQPMO structure indicates which fields are present in the MQPMRs:

```
MQLONG PutMsgRecFields=MQPMRF_MSG_ID + MQPMRF_CORREL_ID;
```

Next, the sample allocates the response records and object records. The object records (MQORs) require at least one pair of names and an even number of names, that is, *ObjectName* and *ObjectQMgrName*.

The next stage involves connecting to the queue managers using MQCONN. The sample attempts to connect to the queue manager associated with the first queue in the MQOR; if this fails, it goes through the object records in turn. You are informed if it is not possible to connect to any queue manager and the program exits.

The target queues are opened using MQOPEN and the message is put to these queues using MQPUT. Any problems and failures are reported in the response records (MQRRs).

Finally, the target queues are closed using MQCLOSE and the program disconnects from the queue manager using MQDISC. The same response records are used for each call stating the *CompCode* and *Reason*.

The Browse sample programs

The Browse sample programs browse messages on a queue using the MQGET call.

See “Features demonstrated in the sample programs” on page 396 for the names of these programs.

UNIX systems and Windows systems

The C version of the program takes 2 parameters

1. The name of the source queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it connects to the default one. For example, enter one of the following:

- `amqsgbr myqueue qmanagername`
- `amqsgbrc myqueue qmanagername`
- `amq0gbr0 myqueue`

where myqueue is the name of the queue that the messages will be viewed from, and qmanagername is the queue manager that owns myqueue.

If you omit the qmanagername, when running the C sample, it assumes that the default queue manager owns the queue.

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target queue
```

Only the first 50 characters of each message are displayed, followed by `- - - truncated` when this is the case.

i5/OS

Each program retrieves copies of all the messages on the queue that you specify when you call the program; the messages remain on the queue.

You can use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You can use the queue SYSTEM.SAMPLE.ALIAS, which is an alias name for the same local queue. The program continues until it reaches the end of the queue or an MQI call fails.

The C samples let you specify the queue manager name, generally as the second parameter, in a similar fashion to the Windows systems samples. For example:

```
CALL PGM(QMQM/AMQSTRG4) PARM('SYSTEM.SAMPLE.TRIGGER' 'QM01')
```

If a queue manager is not specified, it connects to the default one. This is also relevant to the RPG samples. However, with the RPG samples you must supply a queue manager name rather than allowing it to default.

Design of the Browse sample program

The program opens the target queue using the MQOPEN call with the MQOO_BROWSE option. If it cannot open the queue, the program outputs an error message containing the reason code returned by the MQOPEN call.

For each message on the queue, the program uses the MQGET call to copy the message from the queue, then displays the data contained in the message. The MQGET call uses these options:

MQGMO_BROWSE_NEXT
: After the MQOPEN call, the browse cursor is positioned logically before the first message in the queue, so this option causes the ***first*** message to be returned when the call is first made.

MQGMO_NO_WAIT
: The program does not wait if there are no messages on the queue.

MQGMO_ACCEPT_TRUNCATED_MSG
: The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the program displays the truncated message, together with a warning that the message has been truncated.

The program demonstrates how you must clear the *MsgId* and *CorrelId* fields of the MQMD structure after each MQGET call, because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The program continues to the end of the queue; at this point the MQGET call returns the MQRC_NO_MSG_AVAILABLE reason code and the program displays a warning message. If the MQGET call fails, the program displays an error message that contains the reason code.

The program then closes the queue using the MQCLOSE call.

The Browser sample program

The Browser sample program reads and writes both the message descriptor and the message content fields of all the messages on a queue.

The sample program is written as a utility, not just to demonstrate a technique. See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

This program takes these parameters:

1. The name of the source queue
2. The name of the queue manager
3. An optional parameter for properties.

The first two input parameters for this program are mandatory. For example, enter one of the following:

- `amqsbcg myqueue qmanagername`
- `amqsbcgc myqueue qmanagername`

where `myqueue` is the name of the queue on which the messages are going to be browsed, and `qmanagername` is the queue manager that owns `myqueue`.

It reads each message from the queue and writes the following to stdout:

- Formatted message descriptor fields
- Message data (dumped in hex and, where possible, character format)

Permissible values for the property parameter are:

Value	Behavior
0	Default behaviour, as it was for V6. The properties that get delivered to the application depend on the *PropertyControl* queue attribute that the message is retrieved from.

1	A message handle is created and used with the MQGET. Properties of the message, except those contained in the message descriptor (or extension) are displayed in a similar fashion to the message descriptor. For example:
	`****Message properties****`
	`  <property name> : <property value>`
	
	Or if no properties are available:
	`****Message properties****`
	`  None`
	
	Numeric values are displayed using `printf`, string values are surrounding in single quotes, and byte strings are surrounded with X and single quotes, as for the message descriptor.

2	MQGMO_NO_PROPERTIES is specified, so that only message descriptor properties will be returned.

Value	Behavior
3	MQGMO_PROPERTIES_FORCE_MQRFH2 is specified, so that all properties are returned in the message data.
4	MQGMO_PROPERTIES_COMPATIBILITY is specified, so that all properties can be returned depending on whether a version 6 property is included, otherwise the properties are discarded.

The program is restricted to printing the first 65535 characters of the message, and fails with the reason *truncated msg* if a longer message is read.

See the *WebSphere MQ System Administration Guide* for an example of the output from this utility.

The Get sample programs

The Get sample programs get messages from a queue using the MQGET call.

See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

Running the amqsget and amqsgetc samples

These programs each take two parameters:

1. The name of the source queue (required)
2. The name of the queue manager (optional)

If a queue manager is not specified, amqsget connects to the default queue manager, and amqsgetc connects to the queue manager identified by an environment variable or the client channel definition file.

To run these programs, enter one of the following:

- `amqsget myqueue qmanagername`
- `amqsgetc myqueue qmanagername`

where `myqueue` is the name of the queue from which the program will get messages, and `qmanagername` is the queue manager that owns `myqueue`.

If you omit the `qmanagername`, the programs assume the default, or, in the case of the MQI client, the queue manager identified by an environment variable or the client channel definition file.

Running the amq0get sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the source queue
```

Each program removes messages from the queue that you specify when you call the program. You could use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You could use the queue SYSTEM.SAMPLE.ALIAS, which is an alias name for the same local queue. The program continues until the queue is empty or an MQI call fails.

Running the AMQSGET4 and the AMQ0GET4 samples

The Get sample programs get messages from a queue using the MQGET call. The programs are named:

C language	AMQSGET4
COBOL language	AMQ0GET4

Each program removes messages from the queue that you specify when you call the program. You could use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You could use the queue SYSTEM.SAMPLE.ALIAS, which is an alias name for the same local queue. The program continues until the queue is empty or an MQI call fails.

An example of a command to call the C program is:

```
CALL PGM(QMQM/AMQSGET4) PARM('SYSTEM.SAMPLE.LOCAL')
```

Design of the Get sample program

The program opens the target queue using the MQOPEN call with the MQOO_INPUT_AS_Q_DEF option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

For each message on the queue, the program uses the MQGET call to remove the message from the queue, then displays the data contained in the message. The MQGET call uses the MQGMO_WAIT option, specifying a *WaitInterval* of 15 seconds, so that the program waits for this period if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the MQRC_NO_MSG_AVAILABLE reason code.

The program demonstrates how you must clear the *MsgId* and *CorrelId* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the call fails and the program stops.

The program continues until either the MQGET call returns the MQRC_NO_MSG_AVAILABLE reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

The program then closes the queue using the MQCLOSE call.

The Reference Message sample programs

The Reference Message samples allow a large object to be transferred from one node to another (usually on different systems) without the need for the object to be stored on WebSphere MQ queues at either the source or the destination nodes.

A set of sample programs is provided to demonstrate how Reference Messages can be put to a queue, received by message exits, and taken from a queue. The sample programs use Reference Messages to move files. If you want to move other objects

such as databases, or if you want to perform security checks, define your own exit, based on our sample, amqsxrm. The following sections describe the Reference Message sample programs.

There version of the Reference Message exit sample program to use depends on the platform on which the channel is running. On all platforms, use amqsxrma at the sending end. Use amqsxrma at the receiving end if the receiver is running under any WebSphere MQ product except WebSphere MQ for i5/OS; use amqsxrm4 if the receiver is running under WebSphere MQ for i5/OS.

Notes for i5/OS users

To receive a Reference Message using the sample message exit, specify a file in the root file system of IFS or any subdirectory so that a stream file can be created.

The sample message exit on i5/OS creates the file, converts the data to EBCDIC, and sets the code page to your system code page. You can then copy this file to the QSYS.LIB file system using the CPYFRMSTMF command. For example:

```
CPYFRMSTMF FROMSTMF('JANEP/TEST.TXT')
           TOMBR('qsys.lib.janep.lib/test.fie/test.mbr') MBROPT(*REPLACE)
           CVTDTA(*NONE)
```

The CPYFRMSTMF command does not create the file. You must create it before running this command.

If you send a file from QSYS.LIB, no changes are required to the samples. For any other file system ensure that the CCSID specified in the CodedCharSetId field in the MQRMH structure matches the bulk data that you are sending.

When using the integrated file system, create program modules with the SYSIFCOPT(*IFSIO) option set. If you want to move database or fixed-length record files, define your own exit based on the supplied sample AMQSXRM4.

The recommended method of transferring a database file is to convert it to IFS structure, using the CPYTOSTMF command, and then send the Reference Message attaching the IFS file. If you choose to transfer a database file by referring to it from within IFS, but do not convert it to IFS structure, you must specify the member name. Data integrity is not guaranteed if you choose this method.

Running the Reference Message samples

The Reference Message samples run as follows:

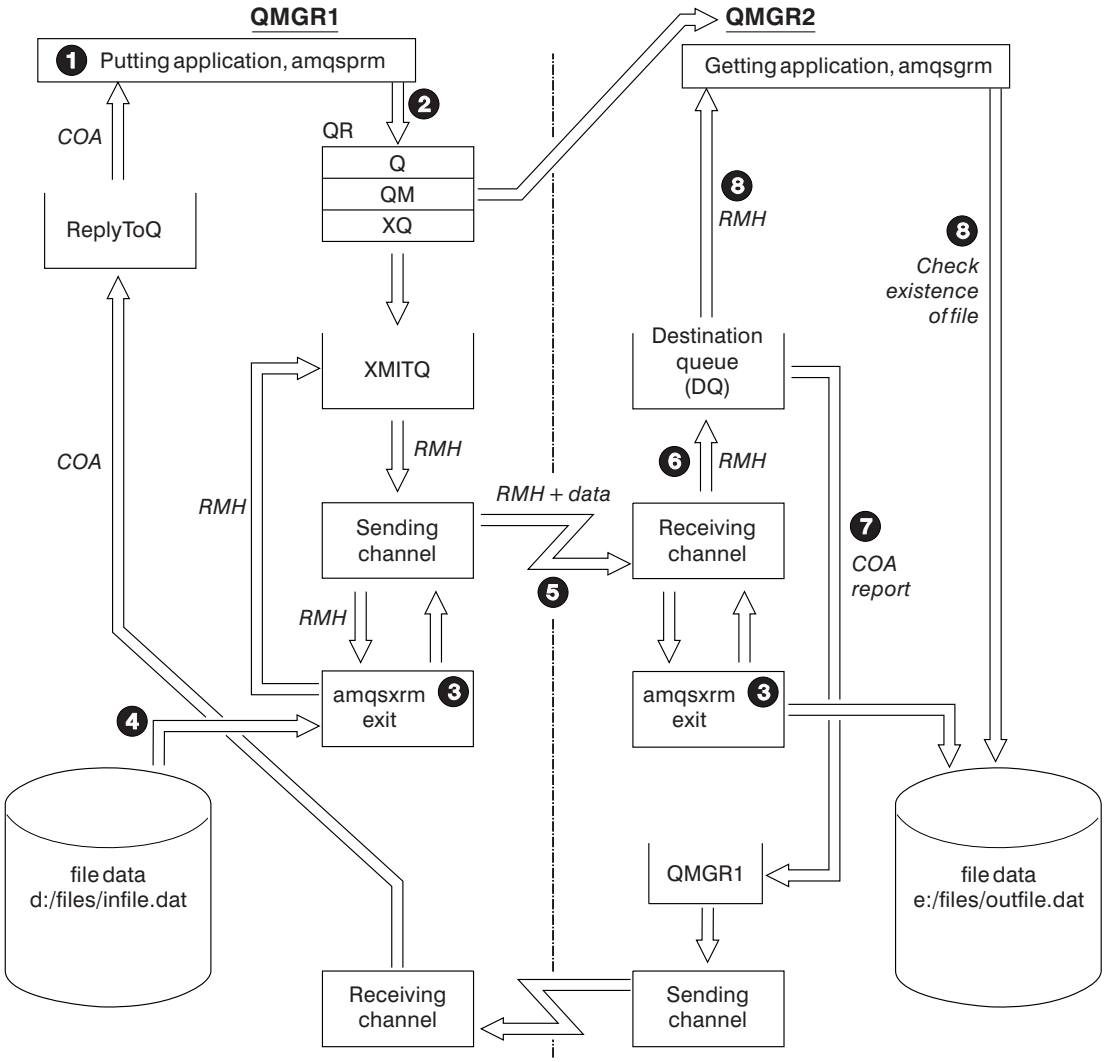


Figure 39. Running the Reference Message samples

1. Set up the environment to start the listeners, channels, and trigger monitors, and define your channels and queues.

   For the purposes of describing how to set up the Reference Message example this refers to the sending machine as MACHINE1 with a queue manager called QMGR1 and the receiving machine as MACHINE2 with a queue manager called QMGR2.

   Note: The following definitions allow a Reference Message to be built to send a file with an object type of FLATFILE from queue manager QMGR1 to QMGR2 and to re-create the file as defined in the call to AMQSPRM (or AMQSPRMA on i5/OS). The Reference Message (including the file data) is sent using channel CHL1 and transmission queue XMITQ and placed on queue DQ. Exception and COA reports are sent back to QMGR1 using the channel REPORT and transmission queue QMGR1.

   The application that receives the Reference Message (AMQSGRM or AMQSGRMA on i5/OS) is triggered using the initiation queue INITQ and process PROC. Ensure that the CONNAME fields are set correctly and the MSGEXIT field reflects your directory structure, depending on machine type and where the WebSphere MQ product is installed.

The MQSC definitions have used an AIX style for defining the exits, so if you are using MQSC on i5/OS you need to modify these accordingly. It is important to note that the message data FLATFILE is case sensitive and the sample will not work unless it is in uppercase.

On machine MACHINE1, queue manager QMGR1

MQSC syntax

```
define chl(chl1) chltype(sdr) trptype(tcp) conname('machine2') xmitq(xmitq)
msgdata(FLATFILE) msgexit('/usr/lpp/mqm/samp/bin/amqsxrm(MsgExit)
')

define ql(xmitq) usage(xmitq)

define chl(report) chltype(rcvr) trptype(tcp) replace

define qr(qr) rname(dq) rqmname(qmgr2) xmitq(xmitq) replace
```

i5/OS command syntax

Note: If you do not specify a queue manager name the system uses the default queue manager.

```
CRTMQMCHL  CHLNAME(CHL1) CHLTYPE(*SDR) MQMNAME(QMGR1) +
             REPLACE(*YES) TRPTYPE(*TCP) +
             CONNAME('MACHINE2(60501)') TMQNAME(XMITQ) +
             MSGEXIT(QMQM/AMQSXRM4) MSGUSRDATA(FLATFILE)

CRTMQMQ    QNAME(XMITQ) QTYPE(*LCL) MQMNAME(QMGR1) +
             REPLACE(*YES) USAGE(*TMQ)

CRTMQMCHL  CHLNAME(REPORT) CHLTYPE(*RCVR) +
               MQMNAME(QMGR1) REPLACE(*YES) TRPTYPE(*TCP)

CRTMQMQ    QNAME(QR) QTYPE(*RMT) MQMNAME(QMGR1) +
             REPLACE(*YES) RMTQNAME(DQ) +
             RMTMQMNAME(QMGR2) TMQNAME(XMITQ)
```

On machine MACHINE2, queue manager QMGR2

MQSC syntax

```
define chl(chl1) chltype(rcvr) trptype(tcp)
msgexit('/usr/lpp/mqm/samp/bin/amqsxrm(MsgExit)')
        msgdata(flatfile)

define chl(report) chltype(sdr) trptype(tcp) conname('MACHINE1')
        xmitq(qmgr1)

define ql(initq)

define ql(qmgr1) usage(xmitq)

define pro(proc) applicid('/usr/lpp/mqm/samp/bin/amqsgrm')

define ql(dq) initq(initq) process(proc) trigger trigtype(first)
```

i5/OS command syntax

Note: If you do not specify a queue manager name the system uses the default queue manager.

```
CRTMQMCHL  CHLNAME(CHL1) CHLTYPE(*RCVR) MQMNAME(QMGR2) +
             REPLACE(*YES) TRPTYPE(*TCP) +
             MSGEXIT(QMQM/AMQSXRM4) MSGUSRDATA(FLATFILE)

CRTMQMCHL  CHLNAME(REPORT) CHLTYPE(*SDR) MQMNAME(QMGR2) +
             REPLACE(*YES) TRPTYPE(*TCP) +
             CONNAME('MACHINE1(60500)') TMQNAME(QMGR1)
```

```
CRTMQMQ    QNAME(INITQ) QTYPE(*LCL) MQMNAME(QMGR2) +
             REPLACE(*YES) USAGE(*NORMAL)

CRTMQMQ    QNAME(QMGR1) QTYPE(*LCL) MQMNAME(QMGR2) +
             REPLACE(*YES) USAGE(*TMQ)

CRTMQMPRC  PRCNAME(PROC) MQMNAME(QMGR2) REPLACE(*YES) +
             APPID('QMQM/AMQSGRM4')

CRTMQMQ    QNAME(DQ) QTYPE(*LCL) MQMNAME(QMGR2) +
             REPLACE(*YES) PRCNAME(PROC) TRGENBL(*YES) +
             INITQNAME(INITQ)
```

2. Once the above WebSphere MQ objects have been created:
   a. Where applicable to the platform, start the listener for the sending and receiving queue managers
   b. Start the channels CHL1 and REPORT
   c. On the receiving queue manager start the trigger monitor for the initiation queue INITQ
3. Invoke the put Reference Message sample program AMQSPRM (AMQSPRMA on i5/OS) from the command line using the following parameters:

-m Name of the local queue manager; this defaults to the default queue manager
-i Name and location of source file
-o Name and location of destination file
-q Name of queue
-g Name of queue manager where the queue, defined in the -q parameter exists This defaults to the queue manager specified in the -m parameter
-t Object type
-w Wait interval, that is, the waiting time for exception and COA reports from the receiving queue manager

For example, to use the sample with the objects defined above you would use the following parameters:

```
-mQMGR1 -iInput File -oOutput File -qQR -tFLATFILE -w120
```

Increasing the waiting time allows time for a large file to be sent across a network before the program putting the messages times out.

```
amqsprm -q QR -m QMGR1 -i d:\x\file.in -o d:\y\file.out -t FLATFILE
```

i5/OS users:

a. Use the following command:

```
CALL       PGM(QMQM/AMQSPRM4) PARM('-mQMGR1' +
             '-i/refmsgs/rmsg1' +
             '-o/refmsgs/rmsgx' '-qQR' +
             '-gQMGR1' '-tFLATFILE' '-w15')
```

This assumes that the original file `rmsg1` is in IFS directory `/refmsgs` and that you want the destination file to be `rmsgx` in IFS directory `/refmsgs` on the target system.

b. Create your own directory using the CRTDIR command rather than using the root directory.
c. When you call the program that puts data, remember that the output file name needs to reflect the IFS naming convention; for instance /TEST/FILENAME creates a file called FILENAME in the directory TEST.

Note: You can use either a forward slash (/) or a dash (-) when specifying parameters.

For example:

```
amqsprm /i d:\files\infile.dat /o e:\files\outfile.dat /q QR
/m QMGR1 /w 30 /t FLATFILE
```

Note: For UNIX platforms, you must use two backslashes (\\) instead of one to denote the destination file directory. Therefore, the above command looks like this:

```
amqsprm -i /files/infile.dat -o e:\\files\\outfile.dat -q QR
-m QMGR1 -w 30 -t FLATFILE
```

Running the put Reference Message program does the following:

- The Reference Message is put to queue QR on queue manager QMGR1.
- The source file and path is `d:\files\infile.dat` and exists on the system where the example command is issued.
- If the queue QR is a remote queue, the Reference Message is sent to another queue manager, on a different system, where a file is created with the name and path `e:\files\outfile.dat`. The contents of this file are the same as the source file.
- amqsprm waits for 30 seconds for a COA report from the destination queue manager.
- The object type is `flatfile`, so the channel used to move messages from the queue QR must specify this in the *MsgData* field.

4. When you define your channels, select the message exit at both the sending and receiving ends to be amqsxrm. This is defined on WebSphere MQ for Windows as follows:

   ```
   msgexit('<pathname>\amqsxrm.dll(MsgExit)')
   ```

   This is defined on WebSphere MQ for AIX, WebSphere MQ for HP-UX, and WebSphere MQ for Solaris as follows:

   ```
   msgexit('<pathname>/amqsxrm(MsgExit)')
   ```

   If you specify a pathname, specify the complete name. If you omit the pathname, it is assumed that the program is in the path specified in the qm.ini file (or, on WebSphere MQ for Windows, the path specified in the registry). This is explained fully in WebSphere MQ Intercommunication.
5. The channel exit reads the Reference Message header and finds the file that it refers to.
6. The channel exit can then segment the file before sending it down the channel along with the header. On WebSphere MQ for AIX, WebSphere MQ for HP-UX, and WebSphere MQ for Solaris, change the group owner of the target directory to 'mqm' so that the sample message exit can create the file in that directory. Also, change the permissions of the target directory to allow mqm group members to write to it. The file data is not stored on the WebSphere MQ queues.
7. When the last segment of the file is processed by the receiving message exit, the Reference Message is put to the destination queue specified by amqsprm. If this queue is triggered (that is, the definition specifies *Trigger*, *InitQ*, and *Process* queue attributes), the program specified by the PROC parameter of the destination queue is triggered. The program to be triggered must be defined in the *ApplId* field of the *Process* attribute.
8. When the Reference Message reaches the destination queue (DQ), a COA report is sent back to the putting application (amqsprm).

9. The Get Reference Message sample, amqsgrm, gets messages from the queue specified in the input trigger message and checks the existence of the file.

Design of the Put Reference Message sample (amqsprma.c, AMQSPRM4)

This sample creates a Reference Message that refers to a file and puts it on a specified queue:

1. The sample connects to a local queue manager using MQCONN.
2. It then opens (MQOPEN) a model queue that is used to receive report messages.
3. The sample builds a Reference Message containing the values required to move the file, for example, the source and destination file names and the object type. As an example, the sample shipped with WebSphere MQ builds a Reference Message to send the file `d:\x\file.in` from `QMGR1` to `QMGR2` and to re-create the file as `d:\y\file.out` using the following parameters:

   ```
   amqsprm -q QR -m QMGR1 -i d:\x\file.in -o d:\y\file.out -t FLATFILE
   ```

   Where `QR` is a remote queue definition that refers to a target queue on `QMGR2`.

   Note: For UNIX platforms, use two backslashes (`\\`) instead of one to denote the destination file directory. Therefore, the above command looks like this:

   ```
   amqsprm -q QR -m QMGR1 -i /x/file.in -o d:\\y\\file.out -t FLATFILE
   ```
4. The Reference Message is put (without any file data) to the queue specified by the /q parameter. If this is a remote queue, the message is put to the corresponding transmission queue.
5. The sample waits, for the duration of time specified in the /w parameter (which defaults to 15 seconds), for COA reports, which, along with exception reports, are sent back to the dynamic queue created on the local queue manager (QMGR1).

Design of the Reference Message Exit sample (amqsxrma.c, AMQSXRM4)

This sample recognizes Reference Messages with an object type that matches the object type in the message exit user data field of the channel definition.

For these messages, the following happens:

- At the sender or server channel, the specified length of data is copied from the specified offset of the specified file into the space remaining in the agent buffer after the Reference Message. If the end of the file is not reached, the Reference Message is put back on the transmission queue after updating the *DataLogicalOffset* field.
- At the requester or receiver channel, if the *DataLogicalOffset* field is zero and the specified file does not exist, it is created. The data following the Reference Message is added to the end of the specified file. If the Reference Message is not the last one for the specified file, it is discarded. Otherwise, it is returned to the channel exit, without the appended data, to be put on the target queue.

For sender and server channels, if the *DataLogicalLength* field in the input Reference Message is zero, the remaining part of the file, from *DataLogicalOffset* to the end of the file, is to be sent along the channel. If it is not zero, only the length specified is sent.

If an error occurs (for example, if the sample cannot open a file), MQCXP.*ExitResponse* is set to MQXCC_SUPPRESS_FUNCTION so that the message being processed is put to the dead-letter queue instead of continuing to the destination queue. A feedback code is returned in MQCXP.*Feedback* and returned to the application that put the message in the *Feedback* field of the message descriptor of a report message. This is because the putting application requested exception reports by setting MQRO_EXCEPTION in the *Report* field of the MQMD.

If the encoding or *CodedCharacterSetId* (CCSID) of the Reference Message is different from that of the queue manager, the Reference Message is converted to the local encoding and CCSID. In our sample, amqsprm, the format of the object is MQFMT_STRING, so amqsxrm converts the object data to the local CCSID at the receiving end before the data is written to the file.

Do not specify the format of the file being transferred as MQFMT_STRING if the file contains multibyte characters (for example, DBCS or Unicode). This is because a multibyte character could be split when the file is segmented at the sending end. To transfer and convert such a file, specify the format as something other than MQFMT_STRING so that the Reference Message exit does not convert it and convert the file at the receiving end when the transfer is complete.

Compiling the Reference Message Exit sample:

To compile amqsxrma, use the following commands:

On AIX

```
xlc_r -q64 -e MsgExit -bE:amqsxrm.exp -bM:SRE -o /var/mqm/exits64/amqsxrma
-I/usr/mqm/inc -L/usr/mqm/lib64 -lmqm_r amqsxrma.c
```

On HP-UX

```
$ c89 +DD64 +z -c -D_HPUX_SOURCE -o amqsxrma.o amqsxrma.c -I/opt/mqm/inc
$ ld -b +noenvvar amqsxrma.o -o /var/mqm/exits64/amqsxrma -L/opt/mqm/lib64
-L/usr/lib/pa20_64 -lmqm_r -lpthread
```

On i5/OS

```
 CRTCMOD MODULE(MYLIB/AMQSXRMA) SRCFILE(QMQMSAMP/QCSRC)
   TERASPACE(*YES *TSIFC)
```

Note:

1. To create your module so that it uses the IFS file system, add the option `SYSIFCOPT(*IFSIO)`
2. To create the program for use with non-threaded channels use the following command: `CRTPGM PGM(MYLIB/AMQSXRMA) BNDSRVPGM(QMQM/LIBMQM)`
3. To create the program for use with threaded channels use the following command: `CRTPGM PGM(MYLIB/AMQSXRMA) BNDSRVPGM(QMQM/LIBMQM_R)`

On Linux

```
$ gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsxrma amqsxrma.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -Wl,-rpath=/opt/mqm/lib64 -Wl,-rpath=/usr/lib64 -lmqm_r
```

On Solaris

```
$ cc -xarch=v9 -mt -G -o /var/mqm/exits64/amqsxrma amqsxrma.c -I/opt/mqm/inc
-L/opt/mqm/lib64 -R/opt/mqm/lib64 -R/usr/lib/64 -lmqm -lmqmcs -lmqmzse -lsocket
-lnsl -ldl
```

On Windows

```
cl amqsxrma.c -o amqsxrm.dll -LD -DEFAULTLIB mqm.lib mqmvx.lib amqsxrm.def
```

For general information about writing and compiling channel exits, see Writing and compiling channel-exit programs.

Design of the Get Reference Message sample (amqsgrma.c, AMQSGRM4)

The program logic is as follows:

1. The sample is triggered and extracts the queue and queue manager names from the input trigger message.
2. It then connects to the specified queue manager using MQCONN and opens the specified queue using MQOPEN.
3. The sample issues MQGET with a wait interval of 15 seconds within a loop to get messages from the queue.
4. If a message is a Reference Message, the sample checks the existence of the file that has been transferred.
5. It then closes the queue and disconnects from the queue manager.

The Request sample programs

The Request sample programs demonstrate client/server processing. The samples are the clients that put request messages on a target server queue that is processed by a server program. They wait for the server program to put a reply message on a reply-to queue.

The Request samples put a series of request messages on the target server queue using the MQPUT call. These messages specify the local queue, SYSTEM.SAMPLE.REPLY as the reply-to queue, which can be a local or remote queue. The programs wait for reply messages, then display them. Replies are sent only if the target server queue is being processed by a server application, or if an application is triggered for that purpose (the Inquire, Set, and Echo sample programs are designed to be triggered). The C sample waits 1 minute (the COBOL sample waits 5 minutes), for the first reply to arrive (to allow time for a server application to be triggered), and 15 seconds for subsequent replies, but both samples can end without getting any replies. See “Features demonstrated in the sample programs” on page 396 for the names of the Request sample programs.

Running the amqsreq0.c, amqsreq, and amqsreqc samples

The C version of the program takes 2 parameters:

1. The name of the target server queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it connects to the default one. For example, enter one of the following:

- `amqsreq myqueue qmanagername`
- `amqsreqc myqueue qmanagername`
- `amq0req0 myqueue`

where `myqueue` is the name of the target server queue, and `qmanagername` is the queue manager that owns `myqueue`.

If you omit the qmanagername, when running the C sample, it assumes that the default queue manager owns the queue.

Running the amq0req0.cbl sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target server queue
```

The program takes its input from StdIn and adds each line to the target server queue, taking each line of text as the content of a request message. The program ends when a null line is read.

Running the AMQSREQ4 sample

The C program creates messages by taking data from stdin (the keyboard) with a blank time terminating input. The program takes up to three parameters: the name of the target queue (required), the queue manager name (optional), and the reply-to queue name (optional). If no queue manager name is specified, the default queue manager is used. If no reply-to queue is specified, the SYSTEM.SAMPLE.REPLY queue is used.

Here is an example of how to call the C sample program, specifying the reply-to queue, but letting the queue manager default:

```
CALL PGM(QMQM/AMQSREQ4) PARM('SYSTEM.SAMPLE.LOCAL' '' 'SYSTEM.SAMPLE.REPLY')
```

Note: Remember that queue names are case sensitive. All the queues created by the sample file create program AMQSAMP4 have names created in uppercase characters.

Running the AMQ0REQ4 sample

The COBOL program creates messages by accepting data from the keyboard. To start the program, call the program and specify the name of your target queue as a parameter. The program accepts input from the keyboard into a buffer and creates a request message for each line of text. The program stops when you enter a blank line at the keyboard.

Running the Request sample using triggering

If the sample is used with triggering and one of the Inquire, Set, or Echo sample programs, the line of input must be the queue name of the queue that you want the triggered program to access.

UNIX systems, and Windows systems:

To run the samples using triggering:

1. Start the trigger monitor program RUNMQTRM in one session (the initiation queue SYSTEM.SAMPLE.TRIGGER is available for you to use).
2. Start the amqsreq program in another session.
3. Make sure that you have defined a target server queue.

   The sample queues available to you to use as the target server queue for the request sample to put messages are:

   - SYSTEM.SAMPLE.INQ - for the Inquire sample program
   - SYSTEM.SAMPLE.SET - for the Set sample program

- SYSTEM.SAMPLE.ECHO - for the Echo sample program

  These queues have a trigger type of FIRST, so if there are already messages on the queues before you run the Request sample, server applications are not triggered by the messages you send.

4. Make sure that you have defined a queue for the Inquire, Set or Echo sample program to use.

This means that the trigger monitor is ready when the request sample sends a message.

Note: The sample process definitions created using RUNMQSC and the amqscos0.tst file trigger the C samples. Change the process definitions in amqscos0.tst and use RUNMQSC with this updated file to use COBOL versions.

Figure 40 demonstrates how to use the Request and Inquire samples together.

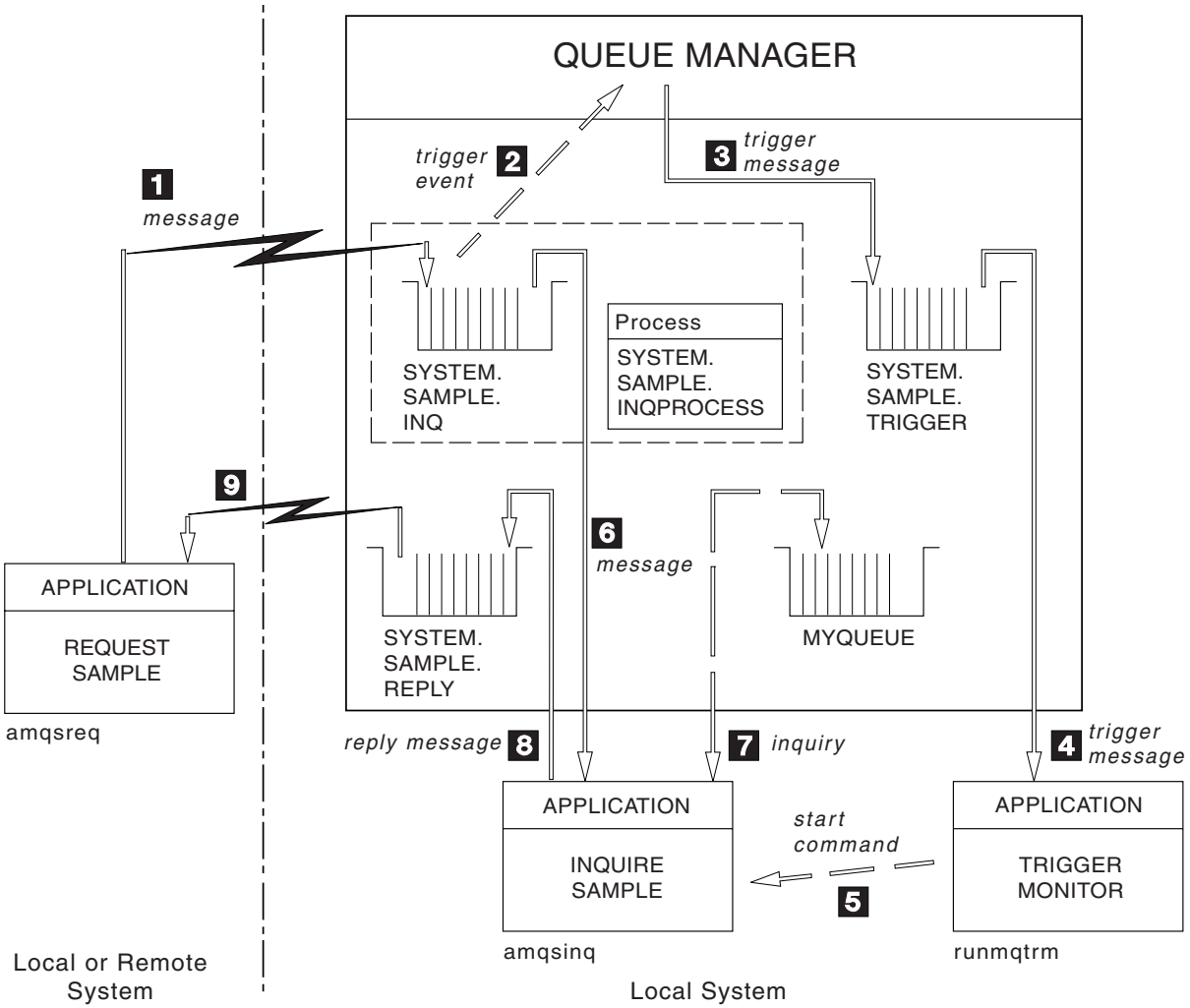


Figure 40. Request and Inquire samples using triggering

In Figure 40 the Request sample puts messages onto the target server queue, SYSTEM.SAMPLE.INQ, and the Inquire sample queries the queue, MYQUEUE.

Alternatively, you can use one of the sample queues defined when you ran amqscos0.tst, or any other queue that you have defined, for the Inquire sample.

Note: The numbers in Figure 40 on page 420 show the sequence of events.

To run the Request and Inquire samples, using triggering:

1. Check that the queues that you want to use are defined. Run amqscos0.tst, to define the sample queues, and define a queue MYQUEUE.
2. Run the trigger monitor command RUNMQTRM:
   ```
   RUNMQTRM -m qmanagername -q SYSTEM.SAMPLE.TRIGGER
   ```
3. Run the request sample
   ```
   amqsreq SYSTEM.SAMPLE.INQ
   ```

   Note: The process object defines what is to be triggered. If the client and server are not running on the same platform, any processes started by the trigger monitor must define *ApplType*, otherwise the server takes its default definitions (that is, the type of application that is normally associated with the server machine) and causes a failure.

   For a list of application types, see the WebSphere MQ Application Programming Reference.
4. Enter the name of the queue that you want the Inquire sample to use:
   ```
   MYQUEUE
   ```
5. Enter a blank line (to end the Request program).
6. The request sample will then display a message, containing the data the Inquire program obtained from MYQUEUE.

You can use more than one queue; in this case, enter the names of the other queues at step 4.

For more information on triggering see "Starting WebSphere MQ applications using triggers" on page 195.

i5/OS:

To try the samples using triggering on i5/OS, start the sample trigger server, AMQSERV4, in one job, then start AMQSREQ4 in another.

This means that the trigger server is ready when the Request sample program sends a message.

Note:

1. The sample definitions created by AMQSAMP4 trigger the C versions of the samples. If you want to trigger the COBOL versions, change the process definitions SYSTEM.SAMPLE.ECHOPROCESS, SYSTEM.SAMPLE.INQPROCESS, and SYSTEM.SAMPLE.SETPROCESS. You can use the CHGMQMPRC command (described in the WebSphere MQ for i5/OS System Administration Guide) to do this, or edit and run your own version of AMQSAMP4.
2. Source code for AMQSERV4 is supplied for the C language only. However, a compiled version (that you can use with the COBOL samples) is supplied in library QMQM.

You could put your request messages on these sample server queues:

- SYSTEM.SAMPLE.ECHO (for the Echo sample programs)
- SYSTEM.SAMPLE.INQ (for the Inquire sample programs)
- SYSTEM.SAMPLE.SET (for the Set sample programs)

A flow chart for the SYSTEM.SAMPLE.ECHO program is shown in Figure 41 on page 423. Using the example data file the command to issue the C program request to this server is:

```
CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMQMSAMP/AMQSDATA(ECHO)')
```

Note: This sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, server applications are not triggered by the messages you send.

If you want to attempt further examples, you can try the following variations:

- Use AMQSTRG4 (or its command line equivalent STRMQMTRM; see the WebSphere MQ for i5/OS System Administration Guide) instead of AMQSERV4 to submit the job instead, but potential job submission delays could make it less easy to follow what is happening.
- Run the SYSTEM.SAMPLE.INQUIRE and SYSTEM.SAMPLE.SET sample programs. Using the example data file the commands to issue the C program requests to these servers are, respectively:

  ```
  CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMQMSAMP/AMQSDATA(INQ)')
  CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMQMSAMP/AMQSDATA(SET)')
  ```

  These sample queues also have a trigger type of FIRST.

Design of the Request sample program

The program opens the target server queue so that it can put messages. It uses the MQOPEN call with the MQOO_OUTPUT option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

The program then opens the reply-to queue called SYSTEM.SAMPLE.REPLY so that it can get reply messages. For this, the program uses the MQOPEN call with the MQOO_INPUT_EXCLUSIVE option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

For each line of input, the program then reads the text into a buffer and uses the MQPUT call to create a request message containing the text of that line. On this call the program uses the MQRO_EXCEPTION_WITH_DATA report option to request that any report messages sent about the request message will include the first 100 bytes of the message data. The program continues until either it reaches the end of the input or the MQPUT call fails.

The program then uses the MQGET call to remove reply messages from the queue, and displays the data contained in the replies. The MQGET call uses the MQGMO_WAIT, MQGMO_CONVERT, and MQGMO_ACCEPT_TRUNCATED options. The *WaitInterval* is 5 minutes in the COBOL version, and 1 minute in the C version, for the first reply (to allow time for a server application to be triggered), and 15 seconds for subsequent replies. The program waits for these periods if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the MQRC_NO_MSG_AVAILABLE reason code. The call also uses the MQGMO_ACCEPT_TRUNCATED_MSG option, so messages longer than the declared buffer size are truncated.

The program demonstrates how to clear the *MsgId* and *CorrelId* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The program continues until either the MQGET call returns the MQRC_NO_MSG_AVAILABLE reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

The program then closes both the target server queue and the reply-to queue using the MQCLOSE call.

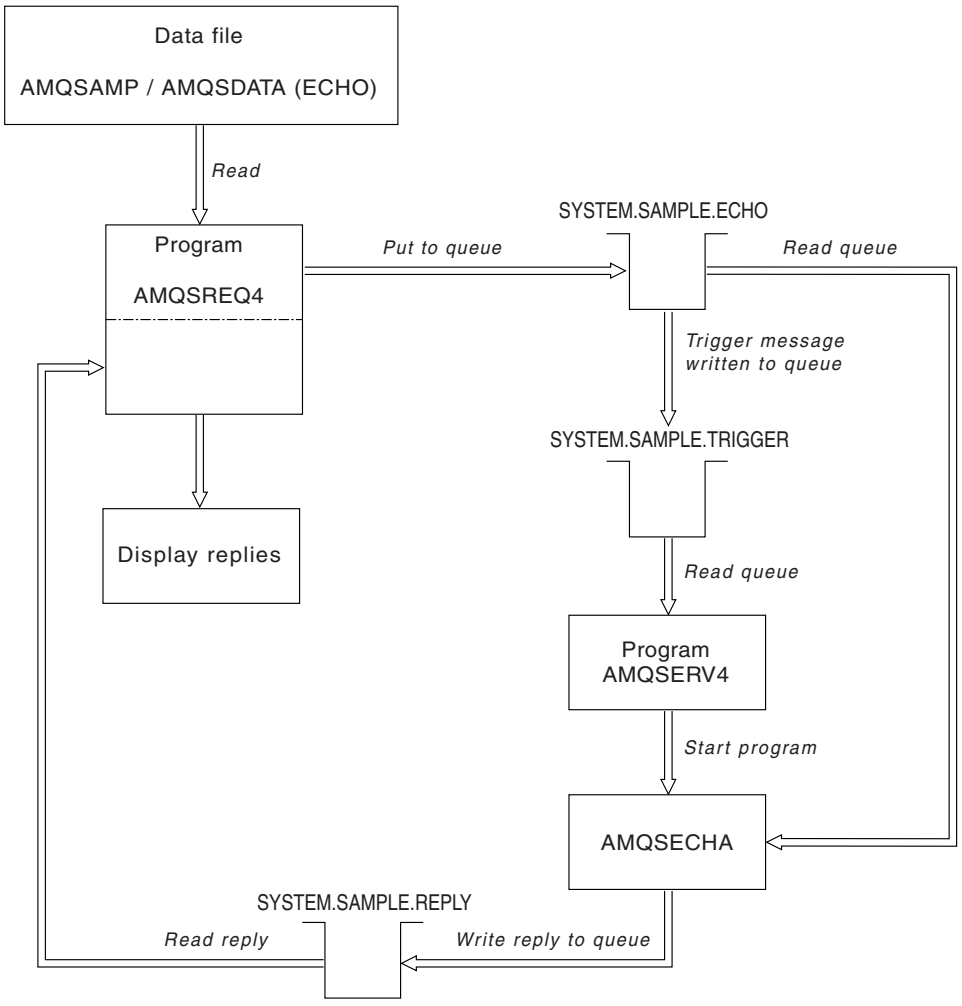


Figure 41. Sample i5/OS Client/Server (Echo) program flowchart

The Inquire sample programs

The Inquire sample programs inquire about some of the attributes of a queue using the MQINQ call.

See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

These programs are intended to run as triggered programs, so their only input is an MQTMC2 (trigger message) structure for i5/OS, Windows systems, and UNIX. This structure contains the name of a target queue whose attributes are to be inquired. The C version also uses the queue manager name. The COBOL version uses the default queue manager.

For the triggering process to work, ensure that the Inquire sample program that you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.INQ. To do this, specify the name of the Inquire sample program that you want to use in the *ApplicId* field of the process definition SYSTEM.SAMPLE.INQPROCESS. For i5/OS, you can use the CHGMQMPRC command described in the WebSphere MQ for i5/OS System Administration Guide for this. The sample queue has a trigger type of FIRST; if there are already messages on the queue before you run the request sample, the inquire sample is not triggered by the messages that you send.

When you have set the definition correctly:

- For UNIX systems and Windows systems, start the **runmqtrm** program in one session, then start the `amqsreq` program in another.
- For i5/OS, start the AMQSERV4 program in one session, then start the AMQSREQ4 program in another. You could use AMQSTRG4 instead of AMQSERV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.INQ. For each request message, the Inquire sample programs send a reply message containing information about the queue specified in the request message. The replies are sent to the reply-to queue specified in the request message.

On i5/OS, if the sample input file member QMQMSAMP.AMQSDATA(INQ) is used, the last queue named does not exist, so the sample returns a report message with a reason code for the failure.

Design of the Inquire sample program

The program opens the queue named in the trigger message structure that it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the MQGMO_ACCEPT_TRUNCATED_MSG and MQGMO_WAIT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data, and opens that queue using the MQOPEN call with the MQOO_INQ option. The program then uses the MQINQ call to inquire about the values of the *InhibitGet*, *CurrentQDepth*, and *OpenInputCount* attributes of the target queue.

If the MQINQ call is successful, the program uses the MQPUT1 call to put a reply message on the reply-to queue. This message contains the values of the three attributes.

If the MQOPEN or MQINQ call is unsuccessful, the program uses the MQPUT1 call to put a report message on the reply-to queue. In the *Feedback* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQINQ call, depending on which one failed.

After the MQINQ call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Set sample programs

The Set sample programs inhibit put operations on a queue by using the MQSET call to change the queue's *InhibitPut* attribute. See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

The programs are intended to run as triggered programs, so their only input is an MQTMC2 (trigger message) structure that contains the name of a target queue whose attributes are to be inquired. The C version also uses the queue manager name. The COBOL version uses the default queue manager.

For the triggering process to work, ensure that the Set sample program that you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.SET. To do this, specify the name of the Set sample program that you want to use in the *ApplicId* field of the process definition SYSTEM.SAMPLE.SETPROCESS. The sample queue has a trigger type of FIRST; if there are already messages on the queue before you run the Request sample, the Set sample is not triggered by the messages that you send.

When you have set the definition correctly:

- For UNIX systems and Windows systems, start the **runmqtrm** program in one session, then start the amqsreq program in another.
- For i5/OS, start the AMQSERV4 program in one session, then start the AMQSREQ4 program in another. You could use AMQSTRG4 instead of AMQSERV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.SET. For each request message, the Set sample programs send a reply message containing a confirmation that put operations have been inhibited on the specified queue. The replies are sent to the reply-to queue specified in the request message.

Design of the Set sample program

The program opens the queue named in the trigger message structure that it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the MQGMO_ACCEPT_TRUNCATED_MSG and MQGMO_WAIT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data and opens that queue using the MQOPEN call with the MQOO_SET option. The program then uses the MQSET call to set the value of the *InhibitPut* attribute of the target queue to MQQA_PUT_INHIBITED.

If the MQSET call is successful, the program uses the MQPUT1 call to put a reply message on the reply-to queue. This message contains the string `PUT inhibited`.

If the MQOPEN or MQSET call is unsuccessful, the program uses the MQPUT1 call to put a `report` message on the reply-to queue. In the *Feedback* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQSET call, depending on which one failed.

After the MQSET call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Echo sample programs

The Echo sample programs echo a message from a message queue to the reply queue.

See “Features demonstrated in the sample programs” on page 396 for the names of these programs.

The programs are intended to run as triggered programs.

On i5/OS, UNIX systems, and Windows systems, their only input is an MQTMC2 (trigger message) structure that contains the name of a target queue and the queue manager. The COBOL version uses the default queue manager.

On i5/OS, for the triggering process to work, ensure that the Echo sample program that you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.ECHO. To do this, specify the name of the Echo sample program that you want to use in the *ApplId* field of the process definition SYSTEM.SAMPLE.ECHOPROCESS. (For this, you can use the CHGMQMPRC command, described in WebSphere MQ for i5/OS System Administration Guide.) The sample queue has a trigger type of FIRST, so, if there are already messages on the queue before you run the Request sample, the Echo sample is not triggered by the messages that you send.

When you have set the definition correctly, first start AMQSERV4 in one job, then start AMQSREQ4 in another. You could use AMQSTRG4 instead of AMQSERV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send messages to queue SYSTEM.SAMPLE.ECHO. The Echo sample programs send a reply message containing the data in the request message to the reply-to queue specified in the request message.

Design of the Echo sample programs

The program opens the queue named in the trigger message structure that it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the MQGMO_ACCEPT_TRUNCATED_MSG, MQGMO_CONVERT, and MQGMO_WAIT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each line of input, the program then reads the text into a buffer and uses the MQPUT1 call to put a request message, containing the text of that line, onto the reply-to queue.

If the MQGET call fails, the program puts a report message on the reply-to queue, setting the *Feedback* field of the message descriptor to the reason code returned by the MQGET.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

On i5/OS, the program can also respond to messages sent to the queue from platforms other than WebSphere MQ for i5/OS, although no sample is supplied for this situation. To make the ECHO program work:

- Write a program, correctly specifying the *Format*, *Encoding*, and *CCSID* parameters, to send text request messages.

  The ECHO program requests the queue manager to perform message data conversion, if this is needed.
- Specify CONVERT(*YES) on the WebSphere MQ for i5/OS sending channel, if the program that you have written does not provide similar conversion for the reply.

The Data-Conversion sample program

The data-conversion sample program is a skeleton of a data conversion exit routine.

See “Features demonstrated in the sample programs” on page 396 for the names of these programs.

Design of the data-conversion sample

Each data-conversion exit routine converts a single named message format. This skeleton is intended as a wrapper for code fragments generated by the data-conversion exit generation utility program.

The utility produces one code fragment for each data structure; several such structures make up a format, so several code fragments are added to this skeleton to produce a routine to do data conversion of the entire format.

The program then checks whether the conversion is a success or failure, and returns the values required to the caller.

The Triggering sample programs

The function provided in the triggering sample is a subset of that provided in the trigger monitor in the **runmqtrm** program.

See "Features demonstrated in the sample programs" on page 396 for the names of these programs.

Running the amqstrg0.c, amqstrg, and amqstrgc samples

The program takes 2 parameters:

1. The name of the initiation queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it connects to the default one. A sample initiation queue will have been defined when you ran amqscos0.tst; the name of that queue is SYSTEM.SAMPLE.TRIGGER, and you can use it when you run this program.

Note: The function in this sample is a subset of the full triggering function that is supplied in the **runmqtrm** program.

Running the AMQSTRG4 sample

This is a trigger monitor for the i5/OS environment. It submits one i5/OS job for each application to be started. This means that there is a processing overhead associated with each trigger message.

AMQSTRG4 (in QCSRC) takes two parameters: the name of the initiation queue that it is to serve, and the name of the queue manager (optional). AMQSAMP4 (in QCLSRC) defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

Using the example trigger queue, the command to issue is:

```
CALL PGM(QMQM/AMQSTRG4) PARM('SYSTEM.SAMPLE.TRIGGER')
```

Alternatively, you can use the CL equivalent STRMQMTRM; see the WebSphere MQ for i5/OS System Administration Guide.

Design of the triggering sample

The triggering sample program opens the initiation queue using the MQOPEN call with the MQOO_INPUT_AS_Q_DEF option. It gets messages from the initiation queue using the MQGET call with the MQGMO_ACCEPT_TRUNCATED_MSG and MQGMO_WAIT options, specifying an unlimited wait interval. The program clears the *MsgId* and *CorrelId* fields before each MQGET call to get messages in sequence.

When it has retrieved a message from the initiation queue, the program tests the message by checking the size of the message to make sure that it is the same size as an MQTM structure. If this test fails, the program displays a warning.

For valid trigger messages, the triggering sample copies data from these fields: *ApplicId*, *EnvrData*, *Version*, and *ApplType*. The last two of these fields are numeric, so the program creates character replacements to use in an MQTMC2 structure for i5/OS, UNIX, and Windows systems.

The triggering sample issues a start command to the application specified in the *ApplicId* field of the trigger message, and passes an MQTMC2 or MQTMC (a character version of the trigger message) structure. In UNIX systems and Windows systems, the *EnvData* field is used as an extension to the invoking command string. In i5/OS, it is used as job submission parameters, for example, the job priority or the job description. See the WebSphere MQ for i5/OS System Administration Guide for a discussion of job priority and job description.

Finally, the program closes the initiation queue.

Running the AMQSERV4 sample

This is a trigger server for the i5/OS environment. For each trigger message, this server runs the start command in its own job to start the specified application. The trigger server can call CICS transactions.

AMQSERV4 takes two parameters: the name of the initiation queue that it is to serve, and the name of the queue manager (optional). AMQSAMP4 defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

Using the example trigger queue the command to issue is:

```
CALL PGM(QMQM/AMQSERV4) PARM('SYSTEM.SAMPLE.TRIGGER')
```

Design of the trigger server

The design of the trigger server is similar to that of the trigger monitor, except that the trigger server:

- Allows MQAT_CICS as well as MQAT_OS400 applications
- Calls i5/OS applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an i5/OS job
- For CICS applications, substitutes the *EnvData*, for example, to specify the CICS region, from the trigger message in the STRCICSUSR command
- Opens the initiation queue for shared input, so that many trigger servers can run at the same time

Note: Programs started by AMQSERV4 must not use the MQDISC call because this stops the trigger server. If programs started by AMQSERV4 use the MQCONN call, they get the MQRC_ALREADY_CONNECTED reason code.

Ending the triggering sample programs on i5/OS

A trigger monitor program can be ended by the sysrequest option 2 (ENDRQS) or by inhibiting gets from the trigger queue.

If the sample trigger queue is used, the command is:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') MQMNAME GETENBL(*NO)
```

Note: Before starting triggering again on this queue, you ***must*** enter the command:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*YES)
```

The Asynchronous Put sample program

The asynchronous put sample program puts messages on a queue using the asynchronous MQPUT call and then retrieves status information using the MQSTAT call. See "Features demonstrated in the sample programs" on page 396 for the name of this program on different platforms.

Running the amqsapt sample

This programs takes up to 6 parameters:

1. The name of the target queue (required)
2. The name of the queue manager (optional)
3. Open options (optional)
4. Close options (optional)
5. The name of the target queue manager (optional)
6. The name of the dynamic queue (optional)

If a queue manager is not specified, amqsapt connects to the default queue manager.

Design of the Asynchronous Put sample program

The program uses the MQOPEN call with the output options supplied, or with the MQOO_OUTPUT and MQOO_FAIL_IF_QUIESCING options to open the target queue for putting messages.

If it cannot open the queue, the program outputs an error message containing the reason code returned by the MQOPEN call. To keep the program simple, on this and on subsequent MQI calls, the program uses default values for many of the options.

For each line of input, the program reads the text into a buffer and uses the MQPUT call with MQPMO_ASYNC_REPONSE to create a datagram message containing the text of that line and asynchronously put it to the target queue. The program continues until it reaches the end of the input or the MQPUT call fails. If the program reaches the end of the input, it closes the queue using the MQCLOSE call.

The program then issues the MQSTAT call, returning an MQSTS structure, and displays messages containing the number of messages put successfully, the number of messages put with a warning, and the number of failures.

Running the samples using remote queues

You can demonstrate remote queuing by running the samples on connected queue managers.

Program amqscos0.tst provides a local definition of a remote queue (SYSTEM.SAMPLE.REMOTE) that uses a remote queue manager named OTHER. To use this sample definition, change OTHER to the name of the second queue manager that you want to use. You must also set up a message channel between your two queue managers; for information on how to do this, see WebSphere MQ Intercommunication.

The Request sample programs put their own local queue manager name in the *ReplyToQMgr* field of messages that they send. The Inquire and Set samples send reply messages to the queue and message queue manager named in the *ReplyToQ* and *ReplyToQMgr* fields of the request messages that they process.

Database coordination samples

Two samples are provided that demonstrate how WebSphere MQ can coordinate both WebSphere MQ updates and database updates within the same unit of work:

1. AMQSXAS0 (in C) or AMQ0XAS0 (in COBOL), which updates a single database within a WebSphere MQ unit of work.
2. AMQSXAG0 (in C) or AMQ0XAG0 (in COBOL), AMQSXAB0 (in C) or AMQ0XAB0 (in COBOL), and AMQSXAF0 (in C) or AMQ0XAF0 (in COBOL), which together update two databases within a WebSphere MQ unit of work, showing how multiple databases can be accessed. These samples are provided to show the use of the MQBEGIN call, mixed SQL and WebSphere MQ calls, and where and when to connect to a database.

Figure 42 shows how the samples provided are used to update databases:

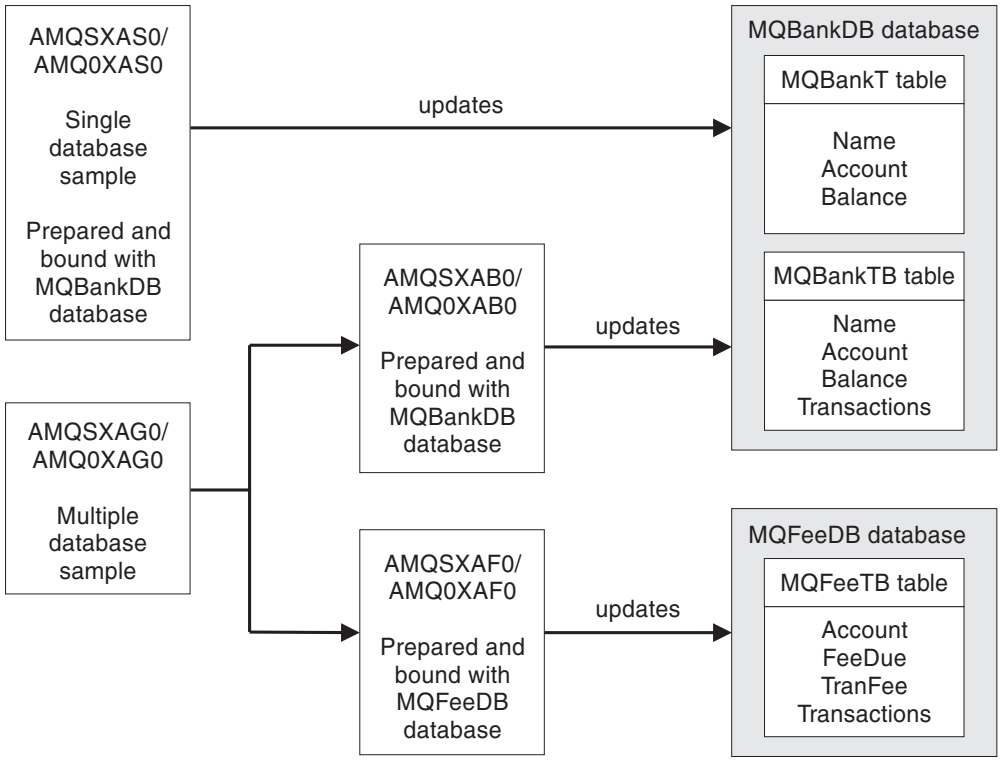


Figure 42. The database coordination samples

The programs read a message from a queue (under syncpoint), then, using the information in the message, obtain the relevant information from the database and update it. The new status of the database is then printed.

The program logic is as follows:

1. Use the name of the input queue from the program argument
2. Connect to the default queue manager (or optionally the supplied name in C) using MQCONN
3. Open a queue (using MQOPEN) for input while there are no failures
4. Start a unit of work using MQBEGIN
5. Get the next message (using MQGET) from the queue under syncpoint
6. Get information from databases
7. Update information from databases
8. Commit changes using MQCMIT

9. Print updated information (no message being available counts as a failure, and the loop ends)
10. Close the queue using MQCLOSE
11. Disconnect from the queue using MQDISC

SQL cursors are used in the samples, so that reads from the databases (that is, multiple instances) are locked while a message is being processed, allowing multiple instances of these programs to run simultaneously. The cursors are explicitly opened, but implicitly closed by the MQCMIT call.

The single database sample (AMQSXAS0 or AMQ0XAS0) has no SQL CONNECT statements and the connection to the database is implicitly made by WebSphere MQ with the MQBEGIN call. The multiple database sample (AMQSXAG0 or AMQ0XAG0, AMQSXAB0 or AMQ0XAB0, and AMQSXAF0 or AMQ0XAF0) has SQL CONNECT statements, as some database products allow only one active connection. If this is not the case for your database product, or if you are accessing a single database in multiple database products, the SQL CONNECT statements can be removed.

The samples are prepared with the IBM DB2 database product, so you might need to modify them to work with other database products.

The SQL error checking uses routines in UTIL.C and CHECKERR.CBL supplied by DB2. These must be compiled or replaced before compiling and linking.

Note: If you are using the Micro Focus COBOL source CHECKERR.MFC for SQL error checking, you must change the program ID to uppercase, that is CHECKERR, for AMQ0XAS0 to link correctly.

Creating the databases and tables

Create the databases and tables before compiling the samples.

To create the databases, use the usual method for your database product, for example:

```
DB2 CREATE DB MQBankDB
DB2 CREATE DB MQFeeDB
```

Create the tables using SQL statements as follows:

In C:

```
EXEC SQL CREATE TABLE MQBankT(Name         VARCHAR(40) NOT NULL,
                              Account      INTEGER     NOT NULL,
                              Balance      INTEGER     NOT NULL,
                              PRIMARY KEY (Account));

EXEC SQL CREATE TABLE MQBankTB(Name         VARCHAR(40) NOT NULL,
                               Account      INTEGER     NOT NULL,
                               Balance      INTEGER     NOT NULL,
                               Transactions INTEGER,
                               PRIMARY KEY (Account));

EXEC SQL CREATE TABLE MQFeeTB(Account      INTEGER     NOT NULL,
                              FeeDue       INTEGER     NOT NULL,
                              TranFee      INTEGER     NOT NULL,
                              Transactions INTEGER,
                              PRIMARY KEY (Account));
```

In COBOL:

```
 EXEC SQL CREATE TABLE
   MQBankT(Name      VARCHAR(40) NOT NULL,
           Account   INTEGER     NOT NULL,
           Balance   INTEGER     NOT NULL,
           PRIMARY KEY (Account))
    END-EXEC.

EXEC SQL CREATE TABLE
  MQBankTB(Name      VARCHAR(40) NOT NULL,
           Account   INTEGER     NOT NULL,
           Balance   INTEGER     NOT NULL,
           Transactions INTEGER,
           PRIMARY KEY (Account))
  END-EXEC.

EXEC SQL CREATE TABLE
  MQFeeTB(Account      INTEGER     NOT NULL,
          FeeDue       INTEGER     NOT NULL,
          TranFee      INTEGER     NOT NULL,
          Transactions INTEGER,
          PRIMARY KEY (Account))
  END-EXEC.
```

Fill in the tables using SQL statements as follows:

```
EXEC SQL INSERT INTO MQBankT VALUES ('Mr Fred Bloggs',1,0);
EXEC SQL INSERT INTO MQBankT VALUES ('Mrs S Smith',2,0);
EXEC SQL INSERT INTO MQBankT VALUES ('Ms Mary Brown',3,0);
:
EXEC SQL INSERT INTO MQBankTB VALUES ('Mr Fred Bloggs',1,0,0);
EXEC SQL INSERT INTO MQBankTB VALUES ('Mrs S Smith',2,0,0);
EXEC SQL INSERT INTO MQBankTB VALUES ('Ms Mary Brown',3,0,0);
:
EXEC SQL INSERT INTO MQFeeTB VALUES (1,0,50,0);
EXEC SQL INSERT INTO MQFeeTB VALUES (2,0,50,0);
EXEC SQL INSERT INTO MQFeeTB VALUES (3,0,50,0);
:
```

Note: For COBOL, use the same SQL statements but add `END_EXEC` at the end of each line.

Precompiling, compiling, and linking the samples

Precompile the .SQC files (in C) and .SQB files (in COBOL), and bind them against the appropriate database to produce the .C or .CBL files. To do this, use the usual method for your database product, as shown below.

Precompiling in C:

```
db2 connect to MQBankDB
db2 prep AMQSXAS0.SQC
db2 connect reset

db2 connect to MQBankDB
db2 prep AMQSXAB0.SQC
db2 connect reset

db2 connect to MQFeeDB
db2 prep AMQSXAF0.SQC
db2 connect reset
```

Precompiling in COBOL:

```
db2 connect to MQBankDB
db2 prep AMQ0XAS0.SQB bindfile target ibmcob
db2 bind AMQ0XAS0.BND
db2 connect reset
```

```
db2 connect to MQBankDB
db2 prep AMQ0XAB0.SQB bindfile target ibmcob
db2 bind AMQ0XAB0.BND
db2 connect reset

db2 connect to MQFeeDB
db2 prep AMQ0XAF0.SQB bindfile target ibmcob
db2 bind AMQ0XAF0.BND
db2 connect reset
```

Compiling and linking:

The following sample commands use the symbol <DB2TOP>. <DB2TOP> represents the installation directory for the DB2 product.

- On AIX the directory path is:

  `/usr/lpp/db2_05_00`
- On HP-UX and Solaris the directory path is:

  `/opt/IBMdb2/V5.0`
- On Windows systems the directory path depends on the path chosen when installing the product. If you chose the default settings the path is:

  `c:\sqllib`

Note: Before issuing the link command on Windows systems, ensure that the LIB environment variable contains paths to the DB2 and WebSphere MQ libraries.

Copy the following files into a temporary directory:

- The `amqsxag0.c` file from your WebSphere MQ installation

  Note: This file can be found in the following directories:
  - On UNIX:

    `<MQMTOP>/samp/xatm`
  - On Windows systems:

    `<MQMTOP>\tools\c\samples\xatm`
- The `.c` files that you have obtained by precompiling the `.sqc` source files, `amqsxas0.sqc`, `amqsxaf0.sqc`, and `amqsxab0.sqc`
- The files `util.c` and `util.h` from your DB2 installation.

  Note: These files can be found in the directory:

  `<DB2TOP>/samples/c`

Build the object files for each `.c` file using the following compiler command for the platform that you are using:

- AIX

  ```
  xlc_r -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
   <FILENAME>.o <FILENAME>.c
  ```
- HP-UX

  ```
  cc -Aa +z -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
   <FILENAME>.o <FILENAME>.c
  ```
- Solaris

  ```
  cc -Aa -KPIC -mt -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
   <FILENAME>.o <FILENAME>.c
  ```
- Windows systems

```
cl /c /I<MQMTOP>\tools\c\include /I<DB2TOP>\include
 <FILENAME>.c
```

Build the amqsxag0 executable using the following link command for the platform that you are using:

- AIX

```
xlc_r -H512 -T512 -L<DB2TOP>/lib -ldb2 -L<MQMTOP>/lib
 -lmqm util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```

- HP-UX Revision 11i

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTOP>/lib -lmqm -lc -lpthread -lcl
 /lib/crt0.o util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```

- Solaris

```
cc -mt -L<DB2TOP>/lib -ldb2 -L<MQMTOP>/lib
 -lmqm -lmqmzse-lmqmcs -lthread -lsocket -lc -lnsl -ldl util.o
  amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```

- Windows systems

```
link util.obj amqsxaf0.obj amqsxab0.obj amqsxag0.obj mqm.lib db2api.lib
 /out:amqsxag0.exe
```

Build the amqsxas0 executable using the following compile and link commands for the platform that you are using:

- AIX

```
xlc_r -H512 -T512 -L<DB2TOP>/lib -ldb2
 -L<MQMTOP>/lib -lmqm util.o amqsxas0.o -o amqsxas0
```

- HP-UX Revision 11i

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTOP>/lib -lmqm -lc -lpthread
 -lcl /lib/crt0.o util.o amqsxas0.o -o amqsxas0
```

- Solaris

```
cc -mt -L<DB2TOP>/lib -ldb2-L<MQMTOP>/lib
 -lqm -lmqmzse -lmqmcs -lthread -lsocket -lc -lnsl -ldl util.o
  amqsxas0.o -o amqsxas0
```

- Windows systems

```
link util.obj amqsxas0.obj mqm.lib db2api.lib /out:amqsxas0.exe
```

Additional information

If you are working on AIX or HP-UX and want to access Oracle, use the xlc_r compiler and link to libmqm_r.a.

Running the samples

Before you run the samples, configure the queue manager with the database product that you are using. For information about how to do this, see the WebSphere MQ System Administration Guide.

C samples:

Messages must be in the following format to be read from a queue:

```
UPDATE Balance change=nnn WHERE Account=nnn
```

AMQSPUT can be used to put the messages on the queue.

The database coordination samples take two parameters:

1. Queue name (required)
2. Queue manager name (optional)

Assuming that you have created and configured a queue manager for the single database sample called singDBQM, with a queue called singDBQ, you increment Mr Fred Bloggs's account by 50 as follows:

```
AMQSPUT singDBQ singDBQM
```

Then key in the following message:

```
UPDATE Balance change=50 WHERE Account=1
```

You can put multiple messages on the queue.

```
AMQSXAS0 singDBQ singDBQM
```

The updated status of Mr Fred Bloggs's account is then printed.

Assuming that you have created and configured a queue manager for the multiple-database sample called multDBQM, with a queue called multDBQ, you decrement Ms Mary Brown's account by 75 as follows:

```
AMQSPUT multDBQ multDBQM
```

Then key in the following message:

```
UPDATE Balance change=-75 WHERE Account=3
```

You can put multiple messages on the queue.

```
AMQSXAG0 multDBQ multDBQM
```

The updated status of Ms Mary Brown's account is then printed.

COBOL samples:

Messages must be in the following format to be read from a queue:

```
UPDATE Balance change=snnnnnnnn WHERE Account=nnnnnnnn
```

For simplicity, the `Balance change` must be a signed eight-character number and the `Account` must be an eight-character number.

The sample AMQSPUT can be used to put the messages on the queue.

The samples take no parameters and use the default queue manager. It can be configured to run only one of the samples at any time. Assuming that you have configured the default queue manager for the single database sample, with a queue called singDBQ, you increment Mr Fred Bloggs's account by 50 as follows:

```
AMQSPUT singDBQ
```

Then key in the following message:

```
UPDATE Balance change=+00000050 WHERE Account=00000001
```

You can put multiple messages on the queue:

```
AMQ0XAS0
```

Type in the name of the queue:

```
singDBQ
```

The updated status of Mr Fred Bloggs's account is then printed.

Assuming that you have configured the default queue manager for the multiple database sample, with a queue called multDBQ, you decrement Ms Mary Brown's account by 75 as follows:

```
AMQSPUT multDBQ
```

Then key in the following message:

```
UPDATE Balance change=-00000075 WHERE Account=00000003
```

You can put multiple messages on the queue:

```
AMQ0XAG0
```

Type in the name of the queue:

```
multDBQ
```

The updated status of Ms Mary Brown's account is then printed.

The CICS transaction sample

A sample CICS transaction program is provided, named amqscic0.ccs for source code and amqscic0 for the executable version. You can build transactions using the standard CICS facilities.

See Chapter 3, "Building a WebSphere MQ application," on page 339 for details on the commands needed for your platform.

The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue, the name of which is contained in the transmission header of the message. Any failures are sent to the queue SYSTEM.SAMPLE.CICS.DLQ.

Note: You can use a sample MQSC script amqscic0.tst to create these queues and sample input queues.

TUXEDO samples

Before running these samples, you must build the server environment.

Note: Throughout this section the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Building the server environment

It is assumed that you have a working TUXEDO environment.

Building the server environment for WebSphere MQ for AIX (32–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /usr/mqm/inc -I /<APPDIR> -L /usr/mqm/lib"
$ export LDOPTS="-lmqm -lmqmcs"
$ export FIELDTBLS=/usr/mqm/samp/amqstxvx.flds
$ export VIEWFILES=/<APPDIR>/amqstxvx.V
$ export LIBPATH=$TUXDIR/lib:/usr/mqm/lib:/lib
```

3. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: -lmqmxa -lmqm -lmqmcs
```

4. Run the commands:

```
$ mkfldhdr    /usr/mqm/samp/amqstxvx.flds
$ viewc       /usr/mqm/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /usr/mqm/samp/amqstxsx.c \
       -f /usr/mqm/lib/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
$ buildserver -o MQSERV2 -f /usr/mqm/samp/amqstxsx.c \
       -f /usr/mqm/lib/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT2:MPUT
       -s MGET2:MGET \
       -v -bshm
$ buildclient -o doputs  -f /usr/mqm/samp/amqstxpx.c \
       -f /usr/mqm/lib/libmqm.a
$ buildclient -o dogets  -f /usr/mqm/samp/amqstxgx.c \
       -f /usr/mqm/lib/libmqm.a
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y /usr/mqm/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for AIX (64–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /usr/mqm/inc -I /<APPDIR> -L /usr/mqm/lib64"
$ export LDOPTS="-lmqm -lmqmcs"
$ export FIELDTBLS=/usr/mqm/samp/amqstxvx.flds
$ export VIEWFILES=/<APPDIR>/amqstxvx.V
$ export LIBPATH=$TUXDIR/lib64:/usr/mqm/lib64:/lib64
```

3. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: -lmqmxa64 -lmqm -lmqmcs
```

4. Run the commands:

```
$ mkfldhdr    /usr/mqm/samp/amqstxvx.flds
$ viewc       /usr/mqm/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /usr/mqm/samp/amqstxsx.c \
       -f /usr/mqm/lib64/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
```

```
$ buildserver -o MQSERV2 -f /usr/mqm/samp/amqstxsx.c \
       -f /usr/mqm/lib64/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT2:MPUT
       -s MGET2:MGET \
       -v -bshm
$ buildclient -o doputs  -f /usr/mqm/samp/amqstxpx.c \
       -f /usr/mqm/lib64/libmqm.a
$ buildclient -o dogets  -f /usr/mqm/samp/amqstxgx.c \
       -f /usr/mqm/lib64/libmqm.a
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y /usr/mqm/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for Solaris (32–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /<APPDIR>"
$ export FIELDTBLS=amqstxvx.flds
$ export VIEWFILES=amqstxvx.V
$ export SHLIB_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
$ export LD_LIBRARY_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
```

3. Add the following to the TUXEDO file udataobj/RM (RM must include /opt/mqm/lib/libmqmcs and /opt/mqm/lib/libmqmzse).

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmxa.a /opt/mqm/lib/libmqm.so \
/opt/tuxedo/lib/libtux.a /opt/mqm/lib/libmqmcs.so  \
/opt/mqm/lib/libmqmzse.so
```

4. Run the commands:

```
$ mkfldhdr    amqstxvx.flds
$ viewc       amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f amqstxsx.c \
       -f /opt/mqm/lib/libmqm.so \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
       -l -ldl
$ buildserver -o MQSERV2 -f amqstxsx.c \
       -f /opt/mqm/lib/libmqm.so \
       -r MQSeries_XA_RMI -s MPUT2:MPUT \
       -s MGET2:MGET \
       -v -bshm
       -l -ldl
$ buildclient -o doputs  -f amqstxpx.c \
```

```
        -f /opt/mqm/lib/libmqm.so  \
        -f /opt/mqm/lib/libmqmzse.co  \
        -f /opt/mqm/lib/libmqmcs.so
$ buildclient -o dogets  -f amqstxgx.c \
        -f /opt/mqm/lib/libmqm.so
        -f /opt/mqm/lib/libmqmzse.co  \
        -f /opt/mqm/lib/libmqmcs.so
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for Solaris (64–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /<APPDIR>"
$ export FIELDTBLS=amqstxvx.flds
$ export VIEWFILES=amqstxvx.V
$ export SHLIB_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib64
$ export LD_LIBRARY_PATH=$TUXDIR/lib64:/opt/mqm/lib64:/lib64
```

3. Add the following to the TUXEDO file udataobj/RM (RM must include /opt/mqm/lib/libmqmcs and /opt/mqm/lib/libmqmzse).

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib64/libmqmxa64.a /opt/mqm/lib64/libmqm.so \
/opt/tuxedo/lib64/libtux.a /opt/mqm/lib64/libmqmcs.so  \
/opt/mqm/lib64/libmqmzse.so
```

4. Run the commands:

```
$ mkfldhdr    amqstxvx.flds
$ viewc       amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f amqstxsx.c \
        -f /opt/mqm/lib64/libmqm.so \
        -r MQSeries_XA_RMI -s MPUT1:MPUT \
        -s MGET1:MGET \
        -v -bshm
        -l -ldl
$ buildserver -o MQSERV2 -f amqstxsx.c \
        -f /opt/mqm/lib64/libmqm.so \
        -r MQSeries_XA_RMI -s MPUT2:MPUT \
        -s MGET2:MGET \
        -v -bshm
        -l -ldl
$ buildclient -o doputs  -f amqstxpx.c \
        -f /opt/mqm/lib64/libmqm.so  \
        -f /opt/mqm/lib64/libmqmzse.co  \
```

```
        -f /opt/mqm/lib64/libmqmcs.so
$ buildclient -o dogets  -f amqstxgx.c \
        -f /opt/mqm/lib64/libmqm.so
        -f /opt/mqm/lib64/libmqmzse.co  \
        -f /opt/mqm/lib64/libmqmcs.so
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for HP-UX (32–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-Aa -D_HPUX_SOURCE"
$ export LDOPTS="-lmqm"
$ export FIELDTBLS=/opt/mqm/samp/amqstxvx.flds
$ export VIEWFILES=<APPDIR>/amqstxvx.V
$ export SHLIB_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
$ export LPATH=$TUXDIR/lib:/opt/mqm/lib:/lib
```

3. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmxa.a /opt/mqm/lib/libmqm.sl \
/opt/tuxedo/lib/libtux.sl
```

4. Run the commands:

```
$ mkfldhdr    /opt/mqm/samp/amqstxvx.flds
$ viewc       /opt/mqm/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /opt/mqm/samp/amqstxsx.c \
        -f /opt/mqm/lib/libmqm.sl \
        -r MQSeries_XA_RMI -s MPUT1:MPUT \
        -s MGET1:MGET \
        -v -bshm
$ buildserver -o MQSERV2 -f /opt/mqm/samp/amqstxsx.c \
        -f /opt/mqm/lib/libmqm.sl \
        -r MQSeries_XA_RMI -s MPUT2:MPUT \
        -s MGET2:MGET \
        -v -bshm
$ buildclient -o doputs  -f /opt/mqm/samp/amqstxpx.c \
        -f /opt/mqm/lib/libmqm.sl
$ buildclient -o dogets  -f /opt/mqm/samp/amqstxgx.c \
        -f /opt/mqm/lib/libmqm.sl
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y /opt/mqm/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for HP-UX (64–bit):

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-Aa -D_HPUX_SOURCE"
$ export LDOPTS="-lmqm"
$ export FIELDTBLS=/opt/mqm/samp/amqstxvx.flds
$ export VIEWFILES=<APPDIR>/amqstxvx.V
$ export SHLIB_PATH=$TUXDIR/lib64:/opt/mqm/lib64:/lib64
$ export LPATH=$TUXDIR/lib64:/opt/mqm/lib64:/lib64
```

3. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmxa64.a /opt/mqm/lib64/libmqm.sl \
/opt/tuxedo/lib64/libtux.sl
```

4. Run the commands:

```
$ mkfldhdr    /opt/mqm/samp/amqstxvx.flds
$ viewc       /opt/mqm/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /opt/mqm/samp/amqstxsx.c \
       -f /opt/mqm/lib64/libmqm.sl \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
$ buildserver -o MQSERV2 -f /opt/mqm/samp/amqstxsx.c \
       -f /opt/mqm/lib64/libmqm.sl \
       -r MQSeries_XA_RMI -s MPUT2:MPUT \
       -s MGET2:MGET \
       -v -bshm
$ buildclient -o doputs  -f /opt/mqm/samp/amqstxpx.c \
       -f /opt/mqm/lib64/libmqm.sl
$ buildclient -o dogets  -f /opt/mqm/samp/amqstxgx.c \
       -f /opt/mqm/li97b/libmqm.sl
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf    -y /opt/mqm/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Building the server environment for WebSphere MQ for Windows (32–bit):

Note: Change the fields identified by <> in the following, to the directory paths:

<MQMDIR>	the directory path specified when WebSphere MQ was installed, for example `g:\Program Files\IBM\WebSphere MQ`
<TUXDIR>	the directory path specified when TUXEDO was installed, for example `f:\tuxedo`
<APPDIR>	the directory path to be used for the sample application, for example `f:\tuxedo\apps\mqapp`

To build the server environment and samples:

1. Create an application directory in which to build the sample application, for example:

   ```
   f:\tuxedo\apps\mqapp
   ```

2. Copy the following sample files from the WebSphere MQ sample directory to the application directory:

   ```
   amqstxmn.mak
   amqstxen.env
   ubbstxcn.cfg
   ```

3. Edit each of these files to set the directory names and directory paths used on your installation.
4. Edit ubbstxcn.cfg (see Figure 43 on page 444) to add details of the machine name and the queue manager that you want to connect to.
5. Add the following line to the TUXEDO file <TUXDIR>udataobj\rm

   ```
   MQSeries_XA_RMI;MQRMIXASwitchDynamic;
      <MQMDIR>\tools\lib\mqmxa.lib <MQMDIR>\tools\lib\mqm.lib
   ```

   where <MQMDIR> is replaced as above. Although shown here as two lines, the new entry must be one line in the file.
6. Set the following environment variables:

   ```
   TUXDIR=<TUXDIR>
   TUXCONFIG=<APPDIR>\tuxconfig
   FIELDTBLS=<MQMDIR>\tools\c\samples\amqstxvx.fld
   LANG=C
   ```

7. Create a TLOG device for TUXEDO. To do this, invoke `tmadmin -c`, and enter the command:

   ```
   crdl -z <APPDIR>\TLOG
   ```

   where *<APPDIR>* is replaced as above.
8. Set the current directory to <APPDIR>, and invoke the sample makefile (amqstxmn.mak) as an external project makefile. For example, with Microsoft Visual C++ , issue the command:

   ```
   msvc amqstxmn.mak
   ```

   Select **build** to build all the sample programs.

```
*RESOURCES
IPCKEY        99999
UID           0
GID           0
MAXACCESSERS  20
MAXSERVERS    20
MAXSERVICES   50
MASTER        SITE1
MODEL         SHM
LDBAL         N

*MACHINES
<MachineName> LMID=SITE1
              TUXDIR="f:\tuxedo"
              APPDIR="f:\tuxedo\apps\mqapp;g:\Program Files\IBM\WebSphere MQ\bin"
              ENVFILE="f:\tuxedo\apps\mqapp\amqstxen.env"
              TUXCONFIG="f:\tuxedo\apps\mqapp\tuxconfig"
              ULOGPFX="f:\tuxedo\apps\mqapp\ULOG"
              TLOGDEVICE="f:\tuxedo\apps\mqapp\TLOG"
              TLOGNAME=TLOG
              TYPE="i386NT"
              UID=0
              GID=0

*GROUPS
GROUP1
        LMID=SITE1  GRPNO=1
        TMSNAME=MQXA
        OPENINFO="MQSeries_XA_RMI:MYQUEUEMANAGER"

*SERVERS
DEFAULT: CLOPT="-A -- -m MYQUEUEMANAGER"

MQSERV1     SRVGRP=GROUP1 SRVID=1
MQSERV2     SRVGRP=GROUP1 SRVID=2

*SERVICES
MPUT1
MGET1
MPUT2
MGET2
```

Figure 43. Example of ubbstxcn.cfg file for WebSphere MQ for Windows

Note: Change the directory names and directory paths to match your installation. Also change the queue manager name MYQUEUEMANAGER to the name of the queue manager that you want to connect to. Other information that you need to add is identified by <> characters.

The sample ubbconfig file for WebSphere MQ for Windows is listed in Figure 43. It is supplied as ubbstxcn.cfg in the WebSphere MQ samples directory.

The sample makefile (see Figure 44 on page 445) supplied for WebSphere MQ for Windows is called ubbstxmn.mak, and is held in the WebSphere MQ samples directory.

```
TUXDIR  = f:\tuxedo
MQMDIR  = g:\Program Files\IBM\WebSphere MQ
APPDIR  = f:\tuxedo\apps\mqapp
MQMLIB  = $(MQMDIR)\tools\lib
MQMINC  = $(MQMDIR)\tools\c\include
MQMSAMP = $(MQMDIR)\tools\c\samples
INC = -f "-I$(MQMINC) -I$(APPDIR)"
DBG = -f "/Zi"

amqstx.exe:
 $(TUXDIR)\bin\mkfldhdr    -d$(APPDIR) $(MQMSAMP)\amqstxvx.fld
 $(TUXDIR)\bin\viewc       -d$(APPDIR) $(MQMSAMP)\amqstxvx.v
 $(TUXDIR)\bin\buildtms    -o MQXA -r MQSeries_XA_RMI
 $(TUXDIR)\bin\buildserver -o MQSERV1 -f $(MQMSAMP)\amqstxsx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
                           -r MQSeries_XA_RMI \
                           -s MPUT1:MPUT -s MGET1:MGET
 $(TUXDIR)\bin\buildserver -o MQSERV2 -f $(MQMSAMP)\amqstxsx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
                           -r MQSeries_XA_RMI \
                           -s MPUT2:MPUT -s MGET2:MGET
 $(TUXDIR)\bin\buildclient -o doputs -f $(MQMSAMP)\amqstxpx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG)
 $(TUXDIR)\bin\buildclient -o dogets -f $(MQMSAMP)\amqstxgx.c \
                           -f $(MQMLIB)\mqm.lib $(INC) -v $(DBG)
 $(TUXDIR)\bin\tmloadcf    -y $(APPDIR)\ubbstxcn.cfg
```

Figure 44. Sample TUXEDO makefile for WebSphere MQ for Windows

Bulding the server environment for WebSphere MQ for Windows (64–bit):

Note: Change the fields identified by <> in the following, to the directory paths:

<MQMDIR>	the directory path specified when WebSphere MQ was installed, for example `g:\Program Files\IBM\WebSphere MQ`
<TUXDIR>	the directory path specified when TUXEDO was installed, for example `f:\tuxedo`
<APPDIR>	the directory path to be used for the sample application, for example `f:\tuxedo\apps\mqapp`

To build the server environment and samples:

1. Create an application directory in which to build the sample application, for example:

   ```
   f:\tuxedo\apps\mqapp
   ```

2. Copy the following sample files from the WebSphere MQ sample directory to the application directory:

   ```
   amqstxmn.mak
   amqstxen.env
   ubbstxcn.cfg
   ```

3. Edit each of these files to set the directory names and directory paths used on your installation.
4. Edit ubbstxcn.cfg (see Figure 45 on page 446) to add details of the machine name and the queue manager that you want to connect to.
5. Add the following line to the TUXEDO file <TUXDIR>udataobj\rm

   ```
   MQSeries_XA_RMI;MQRMIXASwitchDynamic;
      <MQMDIR>\tools\lib64\mqmxa64.lib <MQMDIR>\tools\lib64\mqm.lib
   ```

   where <MQMDIR> is replaced as above. Although shown here as two lines, the new entry must be one line in the file.

6. Set the following environment variables:

   ```
   TUXDIR=<TUXDIR>
   TUXCONFIG=<APPDIR>\tuxconfig
   FIELDTBLS=<MQMDIR>\tools\c\samples\amqstxvx.fld
   LANG=C
   ```

7. Create a TLOG device for TUXEDO. To do this, invoke `tmadmin -c`, and enter the command:

   ```
   crdl -z <APPDIR>\TLOG
   ```

   where <*APPDIR*> is replaced as above.

8. Set the current directory to <APPDIR>, and invoke the sample makefile (amqstxmn.mak) as an external project makefile. For example, with Microsoft Visual C++ , issue the command:

   ```
   msvc amqstxmn.mak
   ```

   Select **build** to build all the sample programs.

```
*RESOURCES
IPCKEY        99999
UID           0
GID           0
MAXACCESSERS  20
MAXSERVERS    20
MAXSERVICES   50
MASTER        SITE1
MODEL         SHM
LDBAL         N

*MACHINES
<MachineName> LMID=SITE1
              TUXDIR="f:\tuxedo"
              APPDIR="f:\tuxedo\apps\mqapp;g:\Program Files\IBM\WebSphere MQ\bin"
              ENVFILE="f:\tuxedo\apps\mqapp\amqstxen.env"
              TUXCONFIG="f:\tuxedo\apps\mqapp\tuxconfig"
              ULOGPFX="f:\tuxedo\apps\mqapp\ULOG"
              TLOGDEVICE="f:\tuxedo\apps\mqapp\TLOG"
              TLOGNAME=TLOG
              TYPE="i386NT"
              UID=0
              GID=0

*GROUPS
GROUP1
        LMID=SITE1  GRPNO=1
        TMSNAME=MQXA
        OPENINFO="MQSeries_XA_RMI:MYQUEUEMANAGER"

*SERVERS
DEFAULT: CLOPT="-A -- -m MYQUEUEMANAGER"

MQSERV1     SRVGRP=GROUP1 SRVID=1
MQSERV2     SRVGRP=GROUP1 SRVID=2

*SERVICES
MPUT1
MGET1
MPUT2
MGET2
```

Figure 45. Example of ubbstxcn.cfg file for WebSphere MQ for Windows

Note: Change the directory names and directory paths to match your installation. Also change the queue manager name MYQUEUEMANAGER to the name of the queue manager that you want to connect to. Other information that you need to add is identified by <> characters.

The sample ubbconfig file for WebSphere MQ for Windows is listed in Figure 45 on page 446. It is supplied as ubbstxcn.cfg in the WebSphere MQ samples directory.

The sample makefile (see Figure 46) supplied for WebSphere MQ for Windows is called ubbstxmn.mak, and is held in the WebSphere MQ samples directory.

```
TUXDIR  = f:\tuxedo
MQMDIR  = g:\Program Files\IBM\WebSphere MQ
APPDIR  = f:\tuxedo\apps\mqapp
MQMLIB  = $(MQMDIR)\tools\lib64
MQMINC  = $(MQMDIR)\tools\c\include
MQMSAMP = $(MQMDIR)\tools\c\samples
INC = -f "-I$(MQMINC) -I$(APPDIR)"
DBG = -f "/Zi"

amqstx.exe:
 $(TUXDIR)\bin\mkfldhdr    -d$(APPDIR) $(MQMSAMP)\amqstxvx.fld
 $(TUXDIR)\bin\viewc       -d$(APPDIR) $(MQMSAMP)\amqstxvx.v
 $(TUXDIR)\bin\buildtms    -o MQXA -r MQSeries_XA_RMI
 $(TUXDIR)\bin\buildserver -o MQSERV1 -f $(MQMSAMP)\amqstxsx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
                           -r MQSeries_XA_RMI \
                           -s MPUT1:MPUT -s MGET1:MGET
 $(TUXDIR)\bin\buildserver -o MQSERV2 -f $(MQMSAMP)\amqstxsx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
                           -r MQSeries_XA_RMI \
                           -s MPUT2:MPUT -s MGET2:MGET
 $(TUXDIR)\bin\buildclient -o doputs -f $(MQMSAMP)\amqstxpx.c \
                           -f $(MQMLIB)\mqm.lib -v $(INC) $(DBG)
 $(TUXDIR)\bin\buildclient -o dogets -f $(MQMSAMP)\amqstxgx.c \
                           -f $(MQMLIB)\mqm.lib $(INC) -v $(DBG)
 $(TUXDIR)\bin\tmloadcf    -y $(APPDIR)\ubbstxcn.cfg
```

Figure 46. Sample TUXEDO makefile for WebSphere MQ for Windows

Server sample program for TUXEDO

This sample server program (amqstxsx) is designed to run with the Put (amqstxpx.c) and the Get (amqstxgx.c) sample programs. The program runs automatically when TUXEDO is started.

Note: You must start your queue manager ***before*** you start TUXEDO.

The sample server provides two TUXEDO services, MPUT1 and MGET1:

- The MPUT1 service is driven by the PUT sample and uses MQPUT1 in syncpoint to put a message in a unit of work controlled by TUXEDO. It takes the parameters QName and Message Text, which are supplied by the PUT sample.
- The MGET1 service opens and closes the queue each time that it gets a message. It takes the parameters QName and Message Text, which are supplied by the GET sample.

Any error messages, reason codes, and status messages are written to the TUXEDO log file.

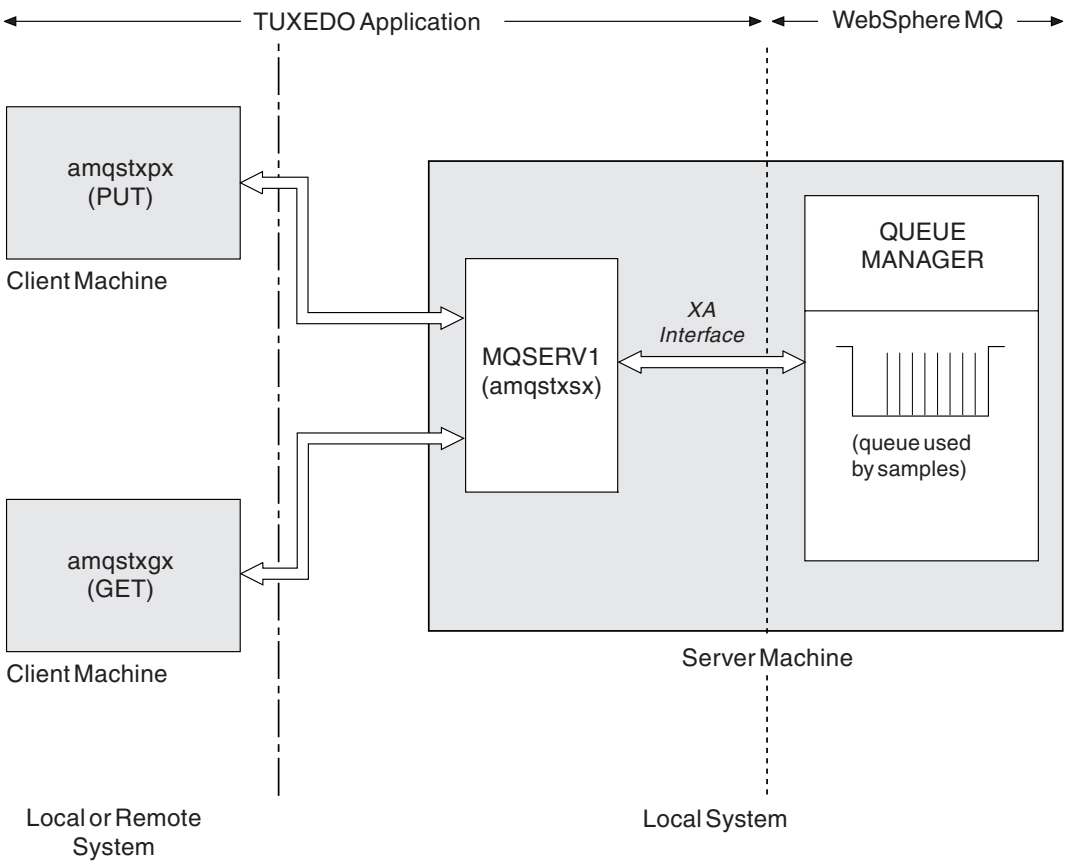


Figure 47. How TUXEDO samples work together

Put sample program for TUXEDO

This sample allows you to put a message on a queue multiple times, in batches, demonstrating syncpointing using TUXEDO as the resource manager.

The sample server program `amqstxsx` must be running for the put sample to succeed; the server sample program connects to the queue manager and uses the XA interface. To run the sample enter:

- `doputs –n queuename –b batchsize –c trancount –t message`

For example:

- `doputs -n myqueue -b 5 -c 6 -t "Hello World"`

This puts 30 messages onto the queue named `myqueue`, in six batches, each with five messages in it. If there are any problems it backs a batch of messages out, otherwise it commits them.

Any error messages are written to the TUXEDO log file and to stderr. Any reason codes are written to stderr.

Get sample for TUXEDO

This sample allows you to get messages from a queue in batches.

The sample server program amqstxsx must be running for the put sample to succeed; the server sample program connects to the queue manager and uses the XA interface. To run the sample enter:

- dogets –n queuename –b batchsize –c trancount

For example:

- dogets -n myqueue -b 6 -c 4

This takes 24 messages off the queue named myqueue, in six batches, each with four messages in it. If you run this after the put example, which puts 30 messages on myqueue, you have only six messages on myqueue. The number of batches and the batch size can vary between putting the messages and getting them.

Any error messages are written to the TUXEDO log file and to stderr. Any reason codes are written to stderr.

Encina sample program

This program puts 10 messages to the queue, backing out the odd numbered messages and committing the even numbered messages. The message is a 4-byte number.

The queue used by this sample is the SYSTEM.DEFAULT.MODEL.QUEUE, so a temporary dynamic queue is created each time that the program is run. You need to run trace to see what happens when the program runs.

Building the AMQSXAE0.C sample

Here, we give the commands needed for Solaris systems as a sample; use the appropriate equivalents for the UNIX system that you are running:

```
cc -xarch=v8plus -mt -o amqsxae amqsxae0.c -I/opt/mqm/inc -I/opt/encina/include \
-L/opt/mqm/lib -L/opt/encina/lib -L/opt/dcelocal/lib \
-R/opt/mqm/lib -R/opt/encina/lib -R/opt/dcelocal/lib -R/usr/lib/32 \
-lmqmxa -lmqz -lmqm -lmqmcs - lmqmzse -lEncina -lEncServer -ldce -lsocket \
-linsl -ldl -lthread -lc -lm
```

Dead-letter queue handler sample

A sample dead-letter queue handler is provided, the name of the executable version is amqsdlq. If you want a dead-letter queue handler that is different from RUNMQDLQ, the source of the sample is available for you to use as your base.

The sample is similar to the dead-letter handler provided within the product but trace and error reporting are different. There are two environment variables available to you:

ODQ_TRACE
Set to YES or yes to switch tracing on

ODQ_MSG
Set to the name of the file containing error and information messages. The file provided is called amqsdlq.msg.

You need to make these variables known to your environment using either the export or set commands, depending on your platform; trace is turned off using the unset command.

You can modify the error message file, amqsdlq.msg, to suit your own requirements. The sample puts messages to stdout, ***not*** to the WebSphere MQ error log file.

The WebSphere MQ System Administration Guide or the *System Management Guide* for your platform explains how the dead-letter handler works, and how you run it.

The Connect sample program

The Connect sample program allows you to explore the MQCONNX call and its options from a client. The sample connects to the queue manager using the MQCONNX call, inquires about the name of the queue manager using the MQINQ call, and displays it.

Note: The Connect sample program is a client sample. You can compile and run it on a server but the function is meaningful only on a client, and only client executables are supplied.

Running the amqscnxc sample

The command-line syntax of the Connect sample program is:

```
amqscnxc [-x ConnName [-c SvrconnChannelName]] [QMgrName]
```

The parameters are optional and their order is not important with the exception of QMgrName, which, if specified, must come last. The parameters are:

ConnName
The TCP/IP connection name of the server queue manager

SvrconnChannelName
The name of the server connection channel

QMgrName
The name of the target queue manager

If you do not specify the TCP/IP connection name, MQCONNX is issued with the *ClientConnPtr* set to NULL. If you specify the TCP/IP connection name but not the server connection channel (the reverse is not allowed), the sample uses the name SYSTEM.DEF.SVRCONN. If you do not specify the target queue manager, the sample connects to whichever queue manager is listening at the given TCP/IP connection name.

Note: If you enter a question mark as the only parameter, or if you enter incorrect parameters, you get a message explaining how to use the program.

If you run the sample with no command-line options, the contents of the MQSERVER environment variable are used to determine the connection information. (In this example MQSERVER is set to `SYSTEM.DEF.SVRCONN/TCP/machine.site.company.com`.) You see output like this:

```
Sample AMQSCNXC start
Connecting to the default queue manager
with no client connection information specified.
Connection established to queue manager machine

Sample AMQSCNXC end
```

If you run the sample and provide a TCP/IP connection name and a server connection channel name but no target queue manager name, like this:

```
amqscnxc -x machine.site.company.com -c SYSTEM.ADMIN.SVRCONN
```

the default queue manager name is used and you see output like this:

```
Sample AMQSCNXC start
Connecting to the default queue manager
using the server connection channel SYSTEM.ADMIN.SVRCONN
on connection name machine.site.company.com.
Connection established to queue manager MACHINE

Sample AMQSCNXC end
```

If you run the sample and provide a TCP/IP connection name and a target queue manager name, like this:

```
amqscnxc -x machine.site.company.com MACHINE
```

you see output like this:

```
Sample AMQSCNXC start
Connecting to queue manager MACHINE
using the server connection channel SYSTEM.DEF.SVRCONN
on connection name machine.site.company.com.
Connection established to queue manager MACHINE

Sample AMQSCNXC end
```

The API exit sample program

The sample API exit generates an MQI trace to a user-specified file with a prefix defined in the MQAPI_TRACE_LOGFILE environment variable. For more information about API exits, see "Using and writing API exits" on page 216.

Source

amqsaxe0.c

Binary

amqsaxe

Configuring for the sample exit

1. Add the following to the qm.ini file (or the registry on Windows). See the WebSphere MQ System Administration Guide for how to do this.

   Platforms other than Windows

   ```
   ApiExitLocal:
      Sequence=100
      Function=EntryPoint
      Module=install_dir/samp/bin/amqsaxe
      Name=SampleApiExit
   ```

   where *install_dir* is the directory where WebSphere MQ was installed.

   Windows

   ```
   ApiExitLocal:
      Sequence=100
      Function=EntryPoint
      Module=install_dir\Tools\c\Samples\bin\amqsaxe
      Name=SampleApiExit
   ```

   where *install_dir* is the directory where WebSphere MQ was installed.

2. Set the environment variable

   ```
   MQAPI_TRACE_LOGFILE=/tmp/MqiTrace
   ```

3. Run your application.

   Output files will appear in the /tmp directory with names like:
   `MqiTrace.<pid>.<tid>.log`

Using the SSPI security exit on Windows systems

This section describes how to use the SSPI channel-exit programs on Windows systems. The supplied exit code is in two formats: object and source.

Object code

The object code file is called amqrspin.dll. For both client and server, it is installed as a standard part of WebSphere MQ for Windows in the exits folder, and is loaded as a standard user exit. You can run the supplied security channel exit and use authentication services in your definition of the channel.

To do this, specify either of the following:

```
SCYEXIT('amqrspin(SCY_KERBEROS)')

SCYEXIT('amqrspin(SCY_NTLM)')
```

To provide support for a restricted channel, specify the following on the SRVCONN channel:

```
SCYDATA('remote_principal_name')
```

where *remote_principal_name* is in the form DOMAIN\user. The secure channel is established only if the name of the remote principal matches *remote_principal_name*.

To use the supplied channel-exit programs between systems that operate within a Kerberos security domain, create a servicePrincipalName for the queue manager.

Source code

The exit source code file is called amqsspin.c. It is in `C:\Program Files\IBM\WebSphere MQ\Tools\c\Samples`.

If you modify the source code, you must recompile the modified source.

You compile and link it in the same way as any other channel exit for the relevant platform, except that SSPI headers need to be accessed at compile time, and the SSPI security libraries, together with any recommended associated libraries, need to be accessed at link time.

Before you execute the following command, make sure that cl.exe, and the Visual C++ library and the `include` folder are available in your path. For example:

```
cl /VERBOSE /LD /MT /I<path_to_Microsoft_platform_SDK\include>
/I<path_to_WebSphere MQ\tools\c\include> amqsspin.c /DSECURITY_WIN32
-link  /DLL /EXPORT:SCY_KERBEROS /EXPORT:SCY_NTLM STACK:8192
```

Note: The source code does not include any provision for tracing or error handling. If you modify and use the source code, add your own tracing and error-handling routines.

Sample programs for WebSphere MQ for z/OS

This chapter describes the sample applications that are delivered with WebSphere MQ for z/OS. These samples demonstrate typical uses of the Message Queue Interface (MQI).

WebSphere MQ for z/OS also provides a sample API-crossing exit program, described in "The API-crossing exit for z/OS" on page 277, and sample data-conversion exits, described in "Writing data-conversion exits" on page 163.

All the sample applications are supplied in source form; several are also supplied in executable form. The source modules include pseudocode that describes the program logic.

Note: Although some of the sample applications have basic panel-driven interfaces, they do not aim to demonstrate how to design the *look and feel* of your applications. For more information on how to design panel-driven interfaces for nonprogrammable terminals, see the *SAA Common User Access: Basic Interface Design Guide* (SC26-4583) and its addendum (GG22-9508). These provide guidelines to help you to design applications that are consistent both within the application and across other applications.

This chapter introduces the sample programs, under these headings:

Features demonstrated in the sample applications

This section summarizes the MQI features demonstrated in each of the sample applications, shows the programming languages in which each sample is written, and the environment in which each sample runs.

Put samples

The Put samples demonstrate how to put messages on a queue using the MQPUT call.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQPUT
- MQCLOSE
- MQDISC

The program is delivered in COBOL and C, and runs in the batch and CICS environment. See Table 34 on page 458 for the batch application and Table 39 on page 462 for the CICS application.

Get samples

The Get samples demonstrate how to get messages from a queue using the MQGET call.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET
- MQCLOSE
- MQDISC

The program is delivered in COBOL and C, and runs in the batch and CICS environment. See Table 34 on page 458 for the batch application and Table 39 on page 462 for the CICS application.

Browse sample

The Browse sample demonstrates how to browse a message, print it, then step through the messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET for browsing messages
- MQCLOSE
- MQDISC

The program is delivered in the COBOL, assembler, PL/I, and C languages. The application runs in the batch environment. See Table 35 on page 458 for the batch application.

Print Message sample

The Print Message sample demonstrates how to remove a message from a queue and print the data in the message, together with all the fields of its message descriptor.

By removing comment characters from two lines in the source module, you can change the program so that it browses, rather than removes, the messages on a queue. This program can usefully be used for diagnosing problems with an application that is putting messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET for removing messages from a queue (with an option to browse)
- MQCLOSE
- MQDISC

The program is delivered in the C language. The application runs in the batch environment. See Table 36 on page 459 for the batch application.

Queue Attributes sample

The Queue Attributes sample demonstrates how to inquire about and set the values of WebSphere MQ for z/OS object attributes.

The application uses these MQI calls:

- MQOPEN
- MQINQ
- MQSET
- MQCLOSE

The program is delivered in the COBOL, assembler, and C languages. The application runs in the CICS environment. See Table 40 on page 462 for the CICS application.

Mail Manager sample

The Mail Manager sample demonstrates these techniques:

- Using alias queues
- Using a model queue to create a temporary dynamic queue
- Using reply-to queues
- Using syncpoints in the CICS and batch environments
- Sending commands to the system-command input queue
- Testing return codes
- Sending messages to remote queue managers, both by using a local definition of a remote queue and by putting messages directly on a named queue at a remote queue manager

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQPUT1
- MQGET
- MQINQ
- MQCMIT
- MQCLOSE
- MQDISC

Three versions of the application are provided:

- A CICS application written in COBOL
- A TSO application written in COBOL
- A TSO application written in C

The TSO applications use the WebSphere MQ for z/OS batch adapter and include some ISPF panels.

See Table 37 on page 460 for the TSO application, and Table 41 on page 463 for the CICS application.

Credit Check sample

The Credit Check sample is a suite of programs that demonstrates these techniques:

- Developing an application that runs in more than one environment
- Using a model queue to create a temporary dynamic queue
- Using a correlation identifier
- Setting and passing context information
- Using message priority and persistence
- Starting programs by using triggering
- Using reply-to queues
- Using alias queues
- Using a dead-letter queue
- Using a namelist
- Testing return codes

The application uses these MQI calls:

- MQOPEN
- MQPUT
- MQPUT1
- MQGET for browsing and getting messages, using the wait and signal options, and for getting a specific message
- MQINQ
- MQSET
- MQCLOSE

The sample can run as a stand-alone CICS application. However, to demonstrate how to design a message queuing application that uses the facilities provided by both the CICS and IMS environments, one module is also supplied as an IMS batch message processing program.

The CICS programs are delivered in C and COBOL. The single IMS program is delivered in C.

See Table 42 on page 463 for the CICS application, and Table 43 on page 465 for the IMS application.

The Message Handler sample

The Message Handler sample allows you to browse, forward, and delete messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQINQ
- MQPUT1
- MQCMIT
- MQBACK
- MQGET
- MQCLOSE

- MQDISC

The program is delivered in C and COBOL programming languages. The application runs under TSO. See Table 38 on page 460 for the TSO application.

Distributed queuing exit samples

The names of the source programs of the distributed queuing exit samples are listed in the following table:

Table 32. Source for the distributed queuing exit samples

Member name	For language	Description	Supplied in library
CSQ4BAX0	Assembler	Source program	SCSQASMS
CSQ4BCX1	C	Source program	SCSQC37S
CSQ4BCX2	C	Source program	SCSQC37S

Note: The source programs are link-edited with CSQXSTUB.

See WebSphere MQ Intercommunication for a description of the distributed queuing exit samples.

Data-conversion exit samples

A skeleton is provided for a data-conversion exit routine, and a sample is shipped with WebSphere MQ illustrating the MQXCNVC call. The names of the source programs of the data-conversion exit samples are listed in the following table:

Table 33. Source for the data conversion exit samples (assembler language only)

Member name	Description	Supplied in library
CSQ4BAX8	Source program	SCSQASMS
CSQ4BAX9	Source program	SCSQASMS
CSQ4CAX9	Source program	SCSQASMS

Note: The source programs are link-edited with CSQASTUB.

See “Writing data-conversion exits” on page 163 for more information.

Preparing and running sample applications for the batch environment

To prepare a sample application that runs in the batch environment, perform the same steps that you would when building any batch WebSphere MQ for z/OS application.

These steps are listed in “Building z/OS batch applications” on page 373.

Alternatively, where we supply an executable form of a sample, you can run it from the thlqual.SCSQLOAD load library.

Note: The assembler language version of the Browse sample uses data control blocks (DCBs), so you must link-edit it using `RMODE(24)`.

The library members to use are listed in Table 34, Table 35, and Table 36 on page 459.

You must edit the run JCL supplied for the samples that you want to use (see Table 34, Table 35, and Table 36 on page 459).

The PARM statement in the supplied JCL contains a number of parameters that you need to modify. To run the C sample programs, separate the parameters by spaces; to run the assembler, COBOL, and PL/I sample programs, separate them by commas. For example, if the name of your queue manager is CSQ1 and you want to run the application with a queue named LOCALQ1, in the COBOL, PL/I, and assembler-language JCL, your PARM statement should look like this:

```
PARM=(CSQ1,LOCALQ1)
```

In the C language JCL, your PARM statement should look like this:

```
PARM=('CSQ1 LOCALQ1')
```

You are now ready to submit the jobs.

Names of the sample batch applications

The names of the programs supplied for each of the sample batch applications, and the libraries where the source, JCL, and, where applicable, the executables reside, are listed in the following tables:

Put and Get samples	Table 34
Browse sample	Table 35
Print message sample	Table 36 on page 459

Table 34. Batch Put and Get samples

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4BCJ1	C	Get source program	SCSQC37S	SCSQLOAD
CSQ4BCK1	C	Put source program	SCSQC37S	SCSQLOAD
CSQ4BVJ1	COBOL	Get source program	SCSQCOBS	SCSQLOAD
CSQ4BVK1	COBOL	Put source program	SCSQCOBS	SCSQLOAD
CSQ4BCJR	C	Sample run JCL	SCSQPROC	None
CSQ4BVJR	COBOL	Sample run JCL	SCSQPROC	None

Table 35. Batch Browse sample

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4BVA1	COBOL	Source program	SCSQCOBS	SCSQLOAD
CSQ4BVAR	COBOL	Sample run JCL	SCSQPROC	None
CSQ4BAA1	Assembler	Source program	SCSQASMS	SCSQLOAD
CSQ4BAAR	Assembler	Sample run JCL	SCSQPROC	None

Table 35. Batch Browse sample (continued)

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4BCA1	C	Source program	SCSQC37S	SCSQLOAD
CSQ4BCAR	C	Sample run JCL	SCSQPROC	None
CSQ4BPA1	PL/I	Source program	SCSQPLIS	SCSQLOAD
CSQ4BPAR	PL/I	Sample run JCL	SCSQPROC	None

Table 36. Batch Print Message sample (C language only)

Member name	Description	Source supplied in library	Executable supplied in library
CSQ4BCG1	Source program	SCSQC37S	SCSQLOAD
CSQ4BCGR	Sample run JCL	SCSQPROC	None
CSQ4BCL1	Browse source program	SCSQC37S	SCSQLOAD
CSQ4BCLR	Sample run JCL	SCSQPROC	None

Preparing sample applications for the TSO environment

To prepare a sample application that runs in the TSO environment, perform the same steps that you would when building any batch WebSphere MQ for z/OS application.

These steps are listed in "Building z/OS batch applications" on page 373. The library members to use are listed in Table 37 on page 460.

Alternatively, where we supply an executable form of a sample, you can run it from the thlqual.SCSQLOAD load library.

For the Mail Manager sample application, ensure that the queues that it uses are available on your system. They are defined in the member **thlqual**.SCSQPROC(CSQ4CVD). To ensure that these queues are always available, you could add these members to your CSQINP2 initialization input data set, or use the CSQUTIL program to load these queue definitions.

Names of the sample TSO applications

The names of the programs supplied for each of the sample TSO applications, and the libraries where the source, JCL, and, for the Message Handler sample only, the executables reside, are listed in the following tables:

Mail manager sample Table 37 on page 460
Message handler sample Table 38 on page 460

These samples use ISPF panels. You must therefore include the ISPF stub, ISPLINK, when you link-edit the programs.

Table 37. TSO Mail Manager sample

Member name	For language	Description	Source supplied in library
CSQ4CVD	independent	WebSphere MQ for z/OS object definitions	SCSQPROC
CSQ40	independent	ISPF messages	SCSQMSGE
CSQ4RVD1	COBOL	CLIST to initiate CSQ4TVD1	SCSQCLST
CSQ4TVD1	COBOL	Source program for Menu program	SCSQCOBS
CSQ4TVD2	COBOL	Source program for Get Mail program	SCSQCOBS
CSQ4TVD4	COBOL	Source program for Send Mail program	SCSQCOBS
CSQ4TVD5	COBOL	Source program for Nickname program	SCSQCOBS
CSQ4VDP1-6	COBOL	Panel definitions	SCSQPNLA
CSQ4VD0	COBOL	Data definition	SCSQCOBC
CSQ4VD1	COBOL	Data definition	SCSQCOBC
CSQ4VD2	COBOL	Data definition	SCSQCOBC
CSQ4VD4	COBOL	Data definition	SCSQCOBC
CSQ4RCD1	C	CLIST to initiate CSQ4TCD1	SCSQCLST
CSQ4TCD1	C	Source program for Menu program	SCSQC37S
CSQ4TCD2	C	Source program for Get Mail program	SCSQC37S
CSQ4TCD4	C	Source program for Send Mail program	SCSQC37S
CSQ4TCD5	C	Source program for Nickname program	SCSQC37S
CSQ4CDP1-6	C	Panel definitions	SCSQPNLA
CSQ4TC0	C	Include file	SCSQC370

Table 38. TSO Message Handler sample

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4TCH0	C	Data definition	SCSQC370	None
CSQ4TCH1	C	Source program	SCSQC37S	SCSQLOAD
CSQ4TCH2	C	Source program	SCSQC37S	SCSQLOAD
CSQ4TCH3	C	Source program	SCSQC37S	SCSQLOAD
CSQ4RCH1	C and COBOL	CLIST to initiate CSQ4TCH1 or CSQ4TVH1	SCSQCLST	None
CSQ4CHP1	C and COBOL	Panel definition	SCSQPNLA	None

Table 38. TSO Message Handler sample (continued)

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4CHP2	C and COBOL	Panel definition	SCSQPNLA	None
CSQ4CHP3	C and COBOL	Panel definition	SCSQPNLA	None
CSQ4CHP9	C and COBOL	Panel definition	SCSQPNLA	None
CSQ4TVH0	COBOL	Data definition	SCSQCOBC	None
CSQ4TVH1	COBOL	Source program	SCSQCOBS	SCSQLOAD
CSQ4TVH2	COBOL	Source program	SCSQCOBS	SCSQLOAD
CSQ4TVH3	COBOL	Source program	SCSQCOBS	SCSQLOAD

Preparing the sample applications for the CICS environment

Before you run the CICS sample programs, log on to CICS using a LOGMODE of 32702. This is because the sample programs have been written to use a 3270 mode 2 screen.

To prepare a sample application that runs in the CICS environment, perform the following steps:

1. Create the symbolic description map and the physical screen map for the sample by assembling the BMS screen definition source (supplied in library **thlqual**.SCSQMAPS, where **thlqual** is the high-level qualifier used by your installation). When you name the maps, use the name of the BMS screen definition source (not available for Put and Get sample programs), but omit the last character of that name.
2. Perform the same steps that you would when building any CICS WebSphere MQ for z/OS application. These steps are listed in "Building CICS applications" on page 375. The library members to use are listed in Table 39 on page 462, Table 40 on page 462, Table 41 on page 463, and Table 42 on page 463.

   Alternatively, where we supply an executable form of a sample, you can run it from the thlqual.SCSQCICS load library.
3. Identify the map set, programs, and transaction to CICS by updating the CICS system definition (CSD) data set. The definitions that you require are in the member **thlqual**.SCSQPROC(CSQ4S100). For guidance on how to do this, see the WebSphere MQ for z/OS System Setup Guide.

   Note: For the Credit Check sample application, you get an error message at this stage if you have not already created the VSAM data set that the sample uses.
4. For the Credit Check and Mail Manager sample applications, ensure that the queues that they use are available on your system. For the Credit Check sample, they are defined in the member **thlqual**.SCSQPROC(CSQ4CVB) for COBOL, and **thlqual**.SCSQPROC(CSQ4CCB) for C. For the Mail Manager sample, they are defined in the member **thlqual**.SCSQPROC(CSQ4CVD). To ensure that these queues are always available, you could add these members to your CSQINP2 initialization input data set, or use the CSQUTIL program to load these queue definitions.

   For the Queue Attributes sample application, you could use one or more of the queues that are supplied for the other sample applications. Alternatively, you

could use your own queues. However, in the form that it is supplied, this sample works only with queues that have the characters CSQ4SAMP in the first eight bytes of their name.

QLOP abend

When the CICS sample applications supplied with WebSphere MQ for z/OS use MQI calls, they do not test for the return codes that indicate that the queue manager is not available. If the queue manager is not available when you attempt to run one of the CICS samples, the sample abends with the CICS abend code QLOP. If this happens, you must connect your queue manager to your CICS system before you attempt to start the sample application again. For information about starting a connection, see the WebSphere MQ for z/OS System Administration Guide.

Names of the sample CICS applications

The names of the programs supplied for each of the sample CICS applications, and the libraries where the source, JCL, and, for the Put, Get, and Queue Attributes samples only, the executables reside, are listed in the following tables:

Put and Get samples	Table 39
Queue attributes sample	Table 40
Mail Manager (CICS) sample	Table 41 on page 463
Credit Check (CICS) sample	Table 42 on page 463

Table 39. CICS Put and Get samples

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4CCK1	C	Source program	SCSQC37S	SCSQCICS
CSQ4CCJ1	C	Source program	SCSQC37S	SCSQCICS
CSQ4CVJ1	COBOL	Source program	SCSQCOBS	SCSQCICS
CSQ4CVK1	COBOL	Source program	SCSQCOBS	SCSQCICS
CSQ4S100	independent	CICS system definition data set	SCSQPROC	None

Table 40. CICS Queue Attributes sample

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4CVC1	COBOL	Source program	SCSQCOBS	SCSQCICS
CSQ4VMSG	COBOL	Message definition	SCSQCOBC	None
CSQ4VCMS	COBOL	BMS screen definition	SCSQMAPS	SCSQCICS (named CSQ4ACM)
CSQ4CAC1	Assembler	Source program	SCSQASMS	SCSQCICS

Table 40. CICS Queue Attributes sample (continued)

Member name	For language	Description	Source supplied in library	Executable supplied in library
CSQ4AMSG	Assembler	Message definition	SCSQMACS	None
CSQ4ACMS	Assembler	BMS screen definition	SCSQMAPS	SCSQCICS (named CSQ4ACM)
CSQ4CCC1	C	Source program	SCSQC37S	SCSQCICS
CSQ4CMSG	C	Message definition	SCSQC370	None
CSQ4CCMS	C	BMS screen definition	SCSQMAPS	SCSQCICS (named CSQ4ACM)
CSQ4S100	independent	CICS system definition data set	SCSQPROC	None

Table 41. CICS Mail Manager sample (COBOL only)

Member name	Description	Source supplied in library
CSQ4CVD	WebSphere MQ for z/OS object definitions	SCSQPROC
CSQ4CVD1	Source for Menu program	SCSQCOBS
CSQ4CVD2	Source for Get Mail program	SCSQCOBS
CSQ4CVD3	Source for Display Message program	SCSQCOBS
CSQ4CVD4	Source for Send Mail program	SCSQCOBS
CSQ4CVD5	Source for Nickname program	SCSQCOBS
CSQ4VDMS	BMS screen definition source	SCSQMAPS
CSQ4S100	CICS system definition data set	SCSQPROC
CSQ4VD0	Data definition	SCSQCOBC
CSQ4VD3	Data definition	SCSQCOBC
CSQ4VD4	Data definition	SCSQCOBC

Table 42. CICS Credit Check sample

Member name	For language	Description	Source supplied in library
CSQ4CVB	independent	WebSphere MQ object definitions	SCSQPROC
CSQ4CCB	independent	WebSphere MQ object definitions	SCSQPROC
CSQ4CVB1	COBOL	Source for user-interface program	SCSQCOBS
CSQ4CVB2	COBOL	Source for credit application manager	SCSQCOBS
CSQ4CVB3	COBOL	Source for checking-account program	SCSQCOBS

Table 42. CICS Credit Check sample (continued)

Member name	For language	Description	Source supplied in library
CSQ4CVB4	COBOL	Source for distribution program	SCSQCOBS
CSQ4CVB5	COBOL	Source for agency-query program	SCSQCOBS
CSQ4CCB1	C	Source for user-interface program	SCSQC37S
CSQ4CCB2	C	Source for credit application manager	SCSQC37S
CSQ4CCB3	C	Source for checking-account program	SCSQC37S
CSQ4CCB4	C	Source for distribution program	SCSQC37S
CSQ4CCB5	C	Source for agency-query program	SCSQC37S
CSQ4CB0	C	Include file	SCSQC370
CSQ4CBMS	C	BMS screen definition source	SCSQMAPS
CSQ4VBMS	COBOL	BMS screen definition source	SCSQMAPS
CSQ4VB0	COBOL	Data definition	SCSQCOBC
CSQ4VB1	COBOL	Data definition	SCSQCOBC
CSQ4VB2	COBOL	Data definition	SCSQCOBC
CSQ4VB3	COBOL	Data definition	SCSQCOBC
CSQ4VB4	COBOL	Data definition	SCSQCOBC
CSQ4VB5	COBOL	Data definition	SCSQCOBC
CSQ4VB6	COBOL	Data definition	SCSQCOBC
CSQ4VB7	COBOL	Data definition	SCSQCOBC
CSQ4VB8	COBOL	Data definition	SCSQCOBC
CSQ4BAQ	independent	Source for VSAM data set	SCSQPROC
CSQ4FILE	independent	JCL to build VSAM data set used by CSQ4CVB3	SCSQPROC
CSQ4S100	independent	CICS system definition data set	SCSQPROC

Preparing the sample application for the IMS environment

Part of the Credit Check sample application can run in the IMS environment.

To prepare this part of the application to run with the CICS sample, first perform the steps described in "Preparing the sample applications for the CICS environment" on page 461.

Then perform the following steps:

1. Perform the same steps that you would when building any IMS WebSphere MQ for z/OS application. These steps are listed in "Building IMS (BMP or MPP) applications" on page 375. The library members to use are listed in Table 43 on page 465.
2. Identify the application program and database to IMS. Samples are provided with PSBGEN, DBDGEN, ACB definition, IMSGEN, and IMSDALOC statements to enable this.

3. Load the database CSQ4CA by tailoring and running the sample JCL provided for this purpose (CSQ4ILDB). This JCL loads the database with data from the file CSQ4BAQ. Update the IMS control region with a DD statement for the database CSQ4CA.
4. Start the checking-account program as a batch message processing (BMP) program by tailoring and running the sample JCL provided for this purpose. This JCL starts a batch-oriented BMP program. To run the program as a message-oriented BMP program, remove the comment characters from the line in the JCL that contains the IN= statement.

Names of the sample IMS application

The source and JCL that are supplied for the Credit Check sample IMS application are listed in Table 43.

Table 43. Source and JCL for the Credit Check IMS sample (C only)

Member name	Description	Supplied in library
CSQ4CVB	WebSphere MQ object definitions	SCSQPROC
CSQ4ICB3	Source for checking-account program	SCSQC37S
CSQ4ICBL	Source for loading the checking-account database	SCSQC37S
CSQ4CBI	Data definition	SCSQC370
CSQ4PSBL	PSBGEN JCL for database-load program	SCSQPROC
CSQ4PSB3	PSBGEN JCL for checking-account program	SCSQPROC
CSQ4DBDS	DBDGEN JCL for database CSQ4CA	SCSQPROC
CSQ4GIMS	IMSGEN macro definitions for CSQ4IVB3 and CSQ4CA	SCSQPROC
CSQ4ACBG	Application control block (ACB) definition for CSQ4IVB3	SCSQPROC
CSQ4BAQ	Source for database	SCSQPROC
CSQ4ILDB	Sample run JCL for database-load job	SCSQPROC
CSQ4ICBR	Sample run JCL for checking-account program	SCSQPROC
CSQ4DYNA	IMSDALOC macro definitions for database	SCSQPROC

The Put samples

The Put sample programs put messages on a queue using the MQPUT call.

The source programs are supplied in C and COBOL in the batch and CICS environments (see Table 34 on page 458 and Table 39 on page 462).

Design of the Put sample

The flow through the program logic is:

1. Connect to the queue manager using the MQCONN call. If this call fails, print the completion and reason codes and stop processing.

   Note: If you are running the sample in a CICS environment, you do not need to issue an MQCONN call; if you do, it returns DEF_HCONN. You can use the connection handle MQHC_DEF_HCONN for the MQI calls that follow.
2. Open the queue using the MQOPEN call with the MQOO_OUTPUT option. On input to this call, the program uses the connection handle that is returned in step 1 on page 468. For the object descriptor structure (MQOD), it uses the default values for all fields except the queue name field, which is passed as a parameter to the program. If the MQOPEN call fails, print the completion and reason codes and stop processing.
3. Create a loop within the program issuing MQPUT calls until the required number of messages are put on the queue. If an MQPUT call fails, the loop is abandoned early, no further MQPUT calls are attempted, and the completion and reason codes are returned.
4. Close the queue using the MQCLOSE call with the object handle returned in step 2 on page 468. If this call fails, print the completion and reason codes.
5. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 1 on page 468. If this call fails, print the completion and reason codes.

   Note: If you are running the sample in a CICS environment, you do not need to issue an MQDISC call.

The Put samples for the batch environment

To run the samples, edit and run the sample JCL, as described in "Preparing and running sample applications for the batch environment" on page 457.

The programs take the following parameters in an EXEC PARM, separated by spaces in C and commas in COBOL:

1. The name of the queue manager (4 characters)
2. The name of the target queue (48 characters)
3. The number of messages (up to 4 digits)
4. The padding character to write in the message (1 character)
5. The number of characters to write in the message (up to 4 digits)
6. The persistence of the message (1 character: P for persistent or N for nonpersistent)

If you enter any of the above parameters wrongly, you receive appropriate error messages.

Any messages from the samples are written to the SYSPRINT data set.

Usage notes:

- To keep the samples simple, there are some minor functional differences between language versions. However, these differences are minimized if you use the layout of the parameters shown in the sample run JCL, CSQ4BCJR, and CSQ4BVJR. None of the differences relate to the MQI.
- CSQ4BCK1 allows you to enter more than four digits for the number of messages sent and the length of the messages.

- For the two numeric fields, enter any digit between 1 and 9999. The value that you enter should be a positive number. For example, to put a single message, you can enter 1, 01, 001, or 0001 as the value. If you enter nonnumeric or negative values, you might receive an error. For example, if you enter -1, the COBOL program sends a one-byte message, but the C program receives an error.
- For both programs, CSQ4BCK1 and CSQ4BVK1, you must enter P in the persistence parameter, ++PER++, if you want the message to be persistent. If you fail to do so, the message will be nonpersistent.

The Put samples for the CICS environment

The transactions take the following parameters separated by commas:

1. The number of messages (up to 4 digits)
2. The padding character to write in the message (1 character)
3. The number of characters to write in the message (up to 4 digits)
4. The persistence of the message (1 character: P for persistent or N for nonpersistent)
5. The name of the target queue (48 characters)

If you enter any of the above parameters wrongly, you receive appropriate error messages.

For the COBOL sample, invoke the Put sample in the CICS environment by entering:

```
MVPT,9999,*,9999,P,QUEUE.NAME
```

For the C sample, invoke the Put sample in the CICS environment by entering:

```
MCPT,9999,*,9999,P,QUEUE.NAME
```

Any messages from the samples are displayed on the screen.

Usage notes:

- To keep the samples simple, there are some minor functional differences between language versions. None of the differences relate to the MQI.
- If you enter a queue name that is longer than 48 characters, its length is truncated to the maximum of 48 characters but no error message is returned.
- Before entering the transaction, press the CLEAR key.
- For the two numeric fields, enter any number between 1 and 9999. The value that you enter should be a positive number. For example, to put a single message, you can enter the value 1, 01, 001, or 0001. If you enter nonnumeric or negative values, you might receive an error. For example, if you enter -1, the COBOL program sends a 1-byte message, and the C program abends with an error from malloc().
- For both programs, CSQ4CCK1 and CSQ4CVK1, enter P in the persistence parameter if you want the message to be persistent. For non-persistent messages, enter N in the persistence parameter. If you enter any other value you receive an error message.
- The messages are put in syncpoint because default values are used for all parameters except those set during program invocation.

The Get samples

The Get sample programs get messages from a queue using the MQGET call.

The source programs are supplied in C and COBOL in the batch and CICS environments (see Table 34 on page 458 and Table 39 on page 462).

Design of the Get sample

The flow through the program logic is:

1. Connect to the queue manager using the MQCONN call. If this call fails, print the completion and reason codes and stop processing.

   Note: If you are running the sample in a CICS environment, you do not need to issue an MQCONN call; if you do, it returns DEF_HCONN. You can use the connection handle MQHC_DEF_HCONN for the MQI calls that follow.
2. Open the queue using the MQOPEN call with the MQOO_INPUT_SHARED and MQOO_BROWSE options. On input to this call, the program uses the connection handle that is returned in step 1. For the object descriptor structure (MQOD), it uses the default values for all fields except the queue name field, which is passed as a parameter to the program. If the MQOPEN call fails, print the completion and reason codes and stop processing.
3. Create a loop within the program issuing MQGET calls until the required number of messages are retrieved from the queue. If an MQGET call fails, the loop is abandoned early, no further MQGET calls are attempted, and the completion and reason codes are returned. The following options are specified on the MQGET call:
   - MQGMO_NO_WAIT
   - MQGMO_ACCEPT_TRUNCATED_MESSAGE
   - MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT
   - MQGMO_BROWSE_FIRST and MQGMO_BROWSE_NEXT

   For a description of these options, see the WebSphere MQ Application Programming Reference. For each message, the message number is printed followed by the length of the message and the message data.
4. Close the queue using the MQCLOSE call with the object handle returned in step 2. If this call fails, print the completion and reason codes.
5. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 1. If this call fails, print the completion and reason codes.

   Note: If you are running the sample in a CICS environment, you do not need to issue an MQDISC call.

The Get samples for the batch environment:

To run the samples, edit and run the sample JCL, as described in "Preparing and running sample applications for the batch environment" on page 457.

The programs take the following parameters in an EXEC PARM, separated by spaces in C and commas in COBOL:

1. The name of the queue manager (4 characters)
2. The name of the target queue (48 characters)
3. The number of messages to get (up to 4 digits)
4. The browse/get message option (1 character: B to browse or D to destructively get the messages)
5. The syncpoint control (1 character: S for syncpoint or N for no syncpoint)

If you enter any of the above parameters incorrectly, you receive appropriate error messages.

Output from the samples is written to the SYSPRINT data set:

```
===================================
PARAMETERS PASSED :
   QMGR        - VC9
   QNAME       - A.Q
   NUMMSGS     - 000000002
   GET         - D
   SYNCPOINT   - N
===================================
MQCONN SUCCESSFUL
MQOPEN SUCCESSFUL
000000000 : 000000010 : **********
000000001 : 000000010 : **********
000000002 MESSAGES GOT FROM QUEUE
MQCLOSE SUCCESSFUL
MQDISC SUCCESSFUL
```

Usage notes:

- To keep the samples simple, there are some minor functional differences between language versions. However, these differences are minimized if you use the layout of the parameters shown in the sample run JCL, CSQ4BCJR, and CSQ4BVJR,. None of the differences relate to the MQI.
- CSQ4BCJ1 allows you to enter more than four digits for the number of messages retrieved.
- Messages longer than 64 KB are truncated.
- CSQ4BCJ1 can only correctly display character messages because it only displays until the first NULL (\0) character is displayed.
- For the numeric number-of-messages field, enter any digit between 1 and 9999. The value that you enter should be a positive number. For example, to get a single message, you can enter 1, 01, 001, or 0001 as the value. If you enter nonnumeric or negative values, you might receive an error. For example, if you enter -1, the COBOL program retrieves one message, but the C program does not retrieve any messages.
- For both programs, CSQ4BCJ1 and CSQ4BVJ1, enter B in the get parameter, ++GET++, if you want to browse the messages.
- For both programs, CSQ4BCJ1 and CSQ4BVJ1, enter S in the syncpoint parameter, ++SYNC++, for messages to be retrieved in syncpoint.

The Get samples for the CICS environment

The transactions take the following parameters in an EXEC PARM, separated by commas:

1. The number of messages to get (up to four digits)
2. The browse/get message option (one character: B to browse or D to destructively get the messages)
3. The syncpoint control (one character: S for syncpoint or N for no syncpoint)
4. The name of the target queue (48 characters)

If you enter any of the above parameters incorrectly, you receive appropriate error messages.

For the COBOL sample, invoke the Get sample in the CICS environment by entering:

```
MVGT,9999,B,S,QUEUE.NAME
```

For the C sample, invoke the Get sample in the CICS environment by entering:

```
MCGT,9999,B,S,QUEUE.NAME
```

When the messages are retrieved from the queue, they are put on a CICS temporary storage queue with the same name as the CICS transaction (for example, MCGT for the C sample).

Here is example output of the Get samples:

```
*************************  TOP OF QUEUE  ***********************
000000000 : 000000010: **********
000000001 : 000000010 :**********
*************************  BOTTOM OF QUEUE  ********************
```

Usage notes:

- To keep the samples simple, there are some minor functional differences between language versions. None of the differences relate to the MQI.
- If you enter a queue name that is longer than 48 characters, its length is truncated to the maximum of 48 characters but no error message is returned.
- Before entering the transaction, press the CLEAR key.
- CSQ4CCJ1 can only correctly display character messages because it only displays until the first NULL (\0) character is displayed.
- For the numeric field, enter any number between 1 and 9999. The value that you enter should be a positive number. For example, to get a single message, you can enter the value 1, 01, 001, or 0001. If you enter a nonnumeric or negative value, you might receive an error.
- Messages longer than 24 526 bytes in C and 9 950 bytes in COBOL are truncated. This is due to the way that the CICS temporary storage queues are used.
- For both programs, CSQ4CCK1 and CSQ4CVK1, enter B in the get parameter if you want to browse the messages, otherwise enter D. This performs destructive MQGET calls. If you enter any other value you receive an error message.
- For both programs, CSQ4CCJ1 and CSQ4CVJ1, enter S in the syncpoint parameter to retrieve messages in syncpoint. If you enter N in the syncpoint parameter, the MQGET calls are issued out of syncpoint. If you enter any other value you receive an error message.

The Browse sample

The Browse sample is a batch application that demonstrates how to browse messages on a queue using the MQGET call.

The application steps through all the messages in a queue, printing the first 80 bytes of each one. You could use this application to look at the messages on a queue without changing them.

Source programs and sample run JCL are supplied in the COBOL, assembler, PL/I, and C languages (see Table 35 on page 458).

To start the application, edit and run the sample run JCL, as described in "Preparing and running sample applications for the batch environment" on page 457. You can look at messages on one of your own queues by specifying the name of the queue in the run JCL.

When you run the application (and there are some messages on the queue), the output data set looks this:

```
07/12/1998                        SAMPLE QUEUE REPORT         PAGE    1
                        QUEUE MANAGER NAME : VC4
                                QUEUE NAME : CSQ4SAMP.DEAD.QUEUE
        RELATIVE
         MESSAGE   MESSAGE
         NUMBER    LENGTH ------------------- MESSAGE DATA -------------

              1       740 HELLO. PLEASE CALL ME WHEN YOU GET BACK.
              2       429 CSQ4BQRM
              3       429 CSQ4BQRM
              4       429 CSQ4BQRM
              5        22 THIS IS A TEST MESSAGE
              6         8 CSQ4TEST
              7        36 CSQ4MSG - ANOTHER TEST MESSAGE.....!
              8         9 CSQ4STOP
                                  ********** END OF REPORT **********
```

If there are no messages on the queue, the data set contains the headings and the *End of report* message only. If an error occurs with any of the MQI calls, the completion and reason codes are added to the output data set.

Design of the Browse sample

The Browse sample application uses a single program module; one is provided in each of the supported programming languages.

The flow through the program logic is:

1. Open a print data set and print the title line of the report. Check that the names of the queue manager and queue have been passed from the run JCL. If both names have been passed, print the lines of the report that contain the names. If they have not, print an error message, close the print data set, and stop processing.

   The way that the program tests the parameters it is passed from the JCL depends on the language in which the program is written; for more information, see "Language-dependent design considerations" on page 472.
2. Connect to the queue manager using the MQCONN call. If this call is not successful, print the completion and reason codes, close the print data set, and stop processing.
3. Open the queue using the MQOPEN call with the MQOO_BROWSE option. On input to this call, the program uses the connection handle returned in step 2. For the object descriptor structure (MQOD), it uses the default values for all the fields except the queue name (which was passed in step 1). If this call is not successful, print the completion and reason codes, close the print data set, and stop processing.
4. Browse the first message on the queue, using the MQGET call. On input to this call, the program specifies:
   - The connection and queue handles from steps 2 and 3
   - An MQMD structure with all fields set to their initial values
   - Two options:
     - MQGMO_BROWSE_FIRST
     - MQGMO_ACCEPT_TRUNCATED_MSG
   - A buffer of size 80 bytes to hold the data copied from the message

   The MQGMO_ACCEPT_TRUNCATED_MSG option allows the call to complete even if the message is longer than the 80-byte buffer specified in the call. If the

message is longer than the buffer, the message is truncated to fit the buffer, and the completion and reason codes are set to show this. The sample was designed so that messages are truncated to 80 characters simply to make the report easy to read. The buffer size is set by a `DEFINE` statement, so you can easily change it if you want to.

5. Perform the following loop until the MQGET call fails:
   a. Print a line of the report showing:
      - The sequence number of the message (this is a count of the browse operations).
      - The true length of the message (not the truncated length). This value is returned in the *DataLength* field of the MQGET call.
      - The first 80 bytes of the message data.
   b. Reset the *MsqId* and *CorrelId* fields of the MQMD structure to nulls
   c. Browse the next message, using the MQGET call with these two options:
      - MQGMO_BROWSE_NEXT
      - MQGMO_ACCEPT_TRUNCATED_MSG
6. If the MQGET call fails, test the reason code to see if the call has failed because the browse cursor has got to the end of the queue. In this case, print the *End of report* message and go to step 7; otherwise, print the completion and reason codes, close the print data set, and stop processing.
7. Close the queue using the MQCLOSE call with the object handle returned in step 3 on page 471.
8. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 2 on page 471.
9. Close the print data set and stop processing.

Language-dependent design considerations

Source modules are provided for the Browse sample in four programming languages.

There are two main differences between the source modules:

- When testing the parameters passed from the run JCL, the COBOL, PL/I, and assembler-language modules search for the comma character (,). If the JCL passes `PARM=(,LOCALQ1)`, the application attempts to open queue LOCALQ1 on the default queue manager. If there is no name after the comma (or no comma), the application returns an error. The C module does not search for the comma character. If the JCL passes a single parameter (for example, `PARM=('LOCALQ1')`), the C module uses this as a queue name on the default queue manager.
- To keep the assembler-language module simple, it uses the date format yy/ddd (for example, 05/116) when it creates the print report. The other modules use the calendar date in mm/dd/yy format.

The Print Message sample

The Print Message sample is a batch application that demonstrates how to remove all the messages from a queue using the MQGET call.

It also prints, for each message, the fields of the message descriptor, followed by the message data. The program prints the data both in hexadecimal and as characters (if they are printable). If a character is not printable, the program replaces it with a period character (.). You can use the program when diagnosing problems with an application that is putting messages on a queue.

You can change the application so that it browses the messages, rather than removing them from the queue. To do this, compile with the option of -DBROWSE, to define the BROWSE macro, as indicated in "Design of the sample" on page 475. Executable code is provided for you in the SCSQLOAD library. Module CSQ4BCG0 is built with -DBROWSE; module CSQ4BCG1 destructively reads the queue.

The application has a single source program, which is written in the C language. Sample run JCL code is also supplied (see Table 36 on page 459).

To start the application, edit and run the sample run JCL, as described in "Preparing and running sample applications for the batch environment" on page 457. When you run the application (and there are some messages on the queue), the output data set looks like that in Figure 48 on page 474.

```
 MQCONN to VC4
 MQOPEN - 'CSQ4SAMP.DEAD.QUEUE'


 MQGET of message number 1
****Message descriptor****
  StrucId  : 'MD  '  Version : 1
  Report   : 0  MsgType : 2
  Expiry   : -1  Feedback : 0
  Encoding : 785  CodedCharSetId : 500
  Format : '        '
  Priority : 3  Persistence : 0
  MsgId : X'C3E2D840E5C3F4404040404040404040A6FE06A95105C620'
  CorrelId : X'C3E2D840E5C3F4404040404040404040A6FE062950C2F125'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'VC4                                           '
  ** Identity Context
  UserIdentifier : 'CICSUSER    '
  Account.Token :
   X'160DD5E3E2D5C5E34BC9C7D7C2F6F1FE060D3B55B60001000000000000000000'
  ApplIdentData  : '                                   '
  ** Origin Context
  PutApplType    : '1'
  PutApplName    : 'VICAUT4 MVB5                '
  PutDate  : '19930203'    PutTime  : '20165982'
  ApplOriginData : '    '
****   Message      ****
 length - 429 bytes

00000000:  C3E2 D8F4 C2D8 D9D4 4040 4040 4040 4040 'CSQ4BQRM        '
00000010:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000020:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000030:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000040:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000050:  4040 4040 4040 40D1 D6C8 D540 D140 4040 '       JOHN J   '
00000060:  4040 4040 4040 4040 4040 40F1 F2F3 F4F5 '           12345'
00000070:  F6F7 F8F9 C6C9 D9E2 E340 C7C1 D3C1 C3E3 '6789FIRST GALACT'
00000080:  C9C3 40C2 C1D5 D240 4040 4040 4040 4040 'IC BANK         '
00000090:  4040 E2D6 D4C5 E3C8 C9D5 C740 C4C9 C6C6 '  SOMETHING DIFF'
000000A0:  C5D9 C5D5 E340 4040 4040 4040 4040 4040 'ERENT           '
000000B0:  F3F5 F0F1 F6F7 F6F2 F1F2 F1F0 F0F0 F0F0 '3501676212100000'
000000C0:  D985 A297 9695 A285 4086 9996 9440 C3E2 'Response from CS'
000000D0:  D8F4 E2C1 D4D7 4BC2 F74B D4C5 E2E2 C1C7 'Q4SAMP.B7.MESSAG'
000000E0:  C5E2 4040 4040 4040 4040 4040 4040 4040 'ES              '
000000F0:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000100:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000110:  4040 4040 40D3 9681 9540 8194 96A4 95A3 '     Loan amount'
00000120:  40F1 F0F0 F0F0 F040 8696 9940 D1D6 C8D5 ' 100000 for JOHN'
00000130:  40D1 4040 4040 4040 4040 4040 4040 4040 ' J              '
00000140:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000150:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
00000160:  4040 4040 C399 8584 89A3 40A6 9699 A388 '    Credit worth'
00000170:  8995 85A2 A240 8995 8485 A740 6040 C2C1 'iness index - BA'
00000180:  C440 4040 4040 4040 4040 4040 4040 4040 'D               '
00000190:  4040 4040 4040 4040 4040 4040 4040 4040 '                '
000001A0:  4040 4040 4040 4040 4040 4040 40        '                '


 No more messages
MQCLOSE
 MQDISC
```

Figure 48. Example of a report from the Print Message sample application

Design of the sample

The Print message sample application uses a single program written in the C language.

The flow through the program logic is:

1. Check that the names of the queue manager and queue have been passed from the run JCL. If they have not, print an error message and stop processing.
2. Connect to the queue manager using the MQCONN call. If this call is not successful, print the completion and reason codes and stop processing; otherwise print the name of the queue manager.
3. Open the queue using the MQOPEN call with the MQOO_INPUT_SHARED option.

   Note: If you want the application to browse the messages rather than remove them from the queue, compile the sample with -DBROWSE, or, add `#define BROWSE` at the top of the source. When you do this, the macro preprocessor adds the line in the program that selects the MQOO_BROWSE option in the compilation.

   On input to this call, the program uses the connection handle returned in step 2. For the object descriptor structure (MQOD), it uses the default values for all the fields except the queue name (which was passed in step 1). If this call is not successful, print the completion and reason codes and stop processing; otherwise, print the name of the queue.
4. Perform the following loop until the MQGET call fails:
   a. Initialize the buffer to blanks so that the message data does not get corrupted by any data already in the buffer.
   b. Set the *MsgId* and *CorrelId* fields of the MQMD structure to nulls so that the MQGET call selects the first message from the queue.
   c. Get a message from the queue, using the MQGET call. On input to this call, the program specifies:
      - The connection and object handles from steps 2 and 3.
      - An MQMD structure with all fields set to their initial values. (*MsgId* and *CorrelId* are reset to nulls for each MQGET call.)
      - The option MQGMO_NO_WAIT.

        Note: If you want the application to browse the messages rather than remove them from the queue, compile the sample with -DBROWSE, or, add `#define BROWSE` at the top of the source. When you do this, the macro preprocessor adds the line in the program that selects the MQGMO_BROWSE_NEXT option to the compilation. When this option is used on a call against a queue for which no browse cursor has previously been used with the current object handle, the browse cursor is positioned logically before the first message.
      - A buffer of size 32 KB to hold the data copied from the message.
   d. Call the printMD subroutine. This prints the name of each field in the message descriptor, followed by its contents.
   e. Print the length of the message, followed by the message data. Each line of message data is in this format:
      - Relative position (in hexadecimal) of this part of the data
      - 16 bytes of hexadecimal data
      - The same 16 bytes of data in character format, if it is printable (nonprintable characters are replaced by periods)

5. If the MQGET call fails, test the reason code to see if the call failed because there are no more messages on the queue. In this case, print the message: No more messages; otherwise, print the completion and reason codes. In both cases, go to step 6.

   Note: The MQGET call fails if it finds a message that has more than 32 KB of data. To change the program to handle larger messages, you could do one of the following:
   - Add the MQGMO_ACCEPT_TRUNCATED_MSG option to the MQGET call, so that the call gets the first 32 KB of data and discards the remainder
   - Make the program leave the message on the queue when it finds one with this amount of data
   - Increase the size of the buffer
6. Close the queue using the MQCLOSE call with the object handle returned in step 3 on page 475.
7. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 2 on page 475.

The Queue Attributes sample

The Queue Attributes sample is a conversational-mode CICS application that demonstrates the use of the MQINQ and MQSET calls.

It shows how to inquire about the values of the *InhibitPut* and *InhibitGet* attributes of queues, and how to change them so that programs cannot put messages on, or get messages from, a queue. You might want to ***lock*** a queue in this way when you are testing a program.

To prevent accidental interference with your own queues, this sample works only on a queue object that has the characters CSQ4SAMP in the first eight bytes of its name. However, the source code includes comments to show you how to remove this restriction.

Source programs are supplied in the COBOL, assembler, and C languages (see Table 40 on page 462).

The assembler-language version of the sample uses reenterable code. To do this, you will notice that the code for each MQI call in that version of the sample includes the MF keyword; for example:

```
CALL MQCONN,(NAME,HCONN,COMPCODE,REASON),MF=(E,PARMAREA),VL
```

(The VL keyword means that you can use the CICS Execution Diagnostic Facility (CEDF) supplied transaction for debugging the program.) For more information on writing reenterable programs, see “Writing reenterable programs” on page 85.

To start the application, start your CICS system and use the following CICS transactions:

- For COBOL, MVC1
- For assembler language, MAC1
- For C, MCC1

You can change the name of any of these transactions by changing the CSD data set mentioned in step 3 on page 461.

Design of the sample

When you start the sample, it displays a screen map that has fields for:

- Name of the queue
- User request (valid actions are: inquire, allow, or inhibit)
- Current status of put operations for the queue
- Current status of get operations for the queue

The first two fields are for user input. The last two fields are filled by the application: they show the word INHIBITED or the word ALLOWED.

The application validates the values that you enter in the first two fields. It checks that the queue name starts with the characters `CSQ4SAMP` and that you entered one of the three valid requests in the Action field. The application converts all your input to uppercase, so you cannot use any queues with names that contain lowercase characters.

If you enter `inquire` in the Action field, the flow through the program logic is:

1. Open the queue using the MQOPEN call with the MQOO_INQUIRE option
2. Call MQINQ using the selectors MQIA_INHIBIT_GET and MQIA_INHIBIT_PUT
3. Close the queue using the MQCLOSE call
4. Analyze the attributes that are returned in the *IntAttrs* parameter of the MQINQ call and move the words INHIBITED or ALLOWED, as appropriate, to the relevant screen fields

If you enter `inhibit` in the Action field, the flow through the program logic is:

1. Open the queue using the MQOPEN call with the MQOO_SET option
2. Call MQSET using the selectors MQIA_INHIBIT_GET and MQIA_INHIBIT_PUT, and with the values MQQA_GET_INHIBITED and MQQA_PUT_INHIBITED in the *IntAttrs* parameter
3. Close the queue using the MQCLOSE call
4. Move the word INHIBITED to the relevant screen fields

If you enter `allow` in the Action field, the application performs similar processing to that for an inhibit request. The only differences are the settings of the attributes and the words displayed on the screen.

When the application opens the queue, it uses the default connection handle to the queue manager. (CICS establishes a connection to the queue manager when you start your CICS system.) The application can trap the following errors at this stage:

- The application is not connected to the queue manager
- The queue does not exist
- The user is not authorized to access the queue
- The application is not authorized to open the queue

For other MQI errors, the application displays the completion and reason codes.

The Mail Manager sample

The Mail Manager sample application is a suite of programs that demonstrates sending and receiving messages, both within a single environment and across

different environments. The application is a simple electronic mailing system that allows users to exchange messages, even if they use different queue managers.

The application demonstrates how to create queues using the MQOPEN call and by putting WebSphere MQ for z/OS commands on the system-command input queue.

Three versions of the application are provided:

- A CICS application written in COBOL
- A TSO application written in COBOL
- A TSO application written in C

Preparing the sample

The Mail Manager is provided in versions that run in two environments. The preparation that you must carry out before you run the application depends on the environment that you want to use.

Users can access mail queues and nickname queues from both TSO and CICS so long as their sign-on user IDs are the same on each system.

Before you can send messages to another queue manager, you must set up a message channel to that queue manager. To do this, use the channel control function of WebSphere MQ, described in WebSphere MQ Intercommunication.

Preparing the sample for the TSO environment:

Follow these steps:

1. Prepare the sample as described in "Preparing sample applications for the TSO environment" on page 459.
2. Tailor the CLIST provided for the sample to define:
   - The location of the panels
   - The location of the message file
   - The location of the load modules
   - The name of the queue manager that you want to use with the application

   A separate CLIST is provided for each language version of the sample:

For the COBOL version:	CSQ4RVD1
For the C version:	CSQ4RCD1

3. Ensure that the queues used by the application are available on the queue manager. (The queues are defined in CSQ4CVD.)

Note: VS COBOL II does not support multitasking with ISPF. This means that you cannot use the Mail Manager sample application on both sides of a split screen. If you do, the results are unpredictable.

Running the sample

To start the sample in the TSO environment, execute your tailored version of the CLIST from the TSO command processor within ISPF.

To start the sample in the CICS Transaction Server for OS/390 environment, run transaction MAIL. If you have not already signed on to CICS, the application prompts you to enter a user ID to which it can send your mail.

When you start the application, it opens your mail queue. If this queue does not already exist, the application creates one for you. Mail queues have names of the form CSQ4SAMP.MAILMGR.*userid*, where *userid* depends on the environment:

In TSO
The user's TSO ID

In CICS
The user's CICS sign-on or the user ID entered by the user when prompted when the Mail Manager started

All parts of the queue names that the Mail Manager uses must be uppercase.

The application then presents a menu panel that has options for:

- Read incoming mail
- Send mail
- Create nickname

The menu panel also shows you how many messages are waiting on your mail queue. Each of the menu options displays a further panel:

Read incoming mail
The Mail Manager displays a list of the messages that are on your mail queue. (Only the first 99 messages on the queue are displayed.) For an example of this panel, see Figure 51 on page 483. When you select a message from this list, the contents of the message are displayed (see Figure 52 on page 484).

Send mail
A panel prompts you to enter:

- The name of the user to whom you want to send a message
- The name of the queue manager that owns their mail queue
- The text of your message

In the user name field you can enter either a user ID or a nickname that you created using the Mail Manager. You can leave the queue manager name field blank if the user's mail queue is owned by the same queue manager that you are using, and you must leave it blank if you entered a nickname in the user name field:

- If you specify only a user name, the program first assumes that the name is a nickname, and sends the message to the object defined by that name. If there is no such nickname, the program attempts to send the message to a local queue of that name.
- If you specify both a user name and a queue manager name, the program sends the message to the mail queue that is defined by those two names.

For example, if you want to send a message to user JONESM on remote queue manager QM12, you could send them a message in either of two ways:

- Use both fields to specify user JONESM at queue manager QM12.
- Define a nickname (for example, MARY) for that user and send them a message by putting MARY in the user name field and nothing in the queue manager name field.

Create nickname
You can define an easy-to-remember name that you can use when you

send a message to another user who you contact frequently. You are prompted to enter the user ID of the other user and the name of the queue manager that owns their mail queue.

Nicknames are queues that have names of the form CSQ4SAMP.MAILMGR.*userid.nickname*, where *userid* is your own user ID and *nickname* is the nickname that you want to use. With names structured in this way, users can each have their own set of nicknames.

The type of queue that the program creates depends on how you fill in the fields of the Create Nickname panel:

- If you specify only a user name, or the queue manager name is the same as that of the queue manager to which the Mail Manager is connected, the program creates an alias queue.
- If you specify both a user name and a queue manager name (and the queue manager is not the one to which the Mail Manager is connected), the program creates a local definition of a remote queue. The program does not check the existence of the queue to which this definition resolves, or even that the remote queue manager exists.

For example, if your own user ID is SMITHK and you create a nickname called MARY for user JONESM (who uses the remote queue manager QM12), the nickname program creates a local definition of a remote queue named CSQ4SAMP.MAILMGR.SMITHK.MARY. This definition resolves to Mary's mail queue, which is CSQ4SAMP.MAILMGR.JONESM at queue manager QM12. If you are using queue manager QM12 yourself, the program instead creates an alias queue of the same name (CSQ4SAMP.MAILMGR.SMITHK.MARY).

The C version of the TSO application makes greater use of ISPF's message-handling capabilities than does the COBOL version. You might notice that different error messages are displayed by the C and COBOL versions.

Design of the sample

The following sections describe each of the programs that comprise the Mail Manager sample application.

The relationships between the programs and the panels that the application uses is shown in Figure 49 on page 481 for the TSO version, and Figure 50 on page 482 for the CICS Transaction Server for OS/390 version.

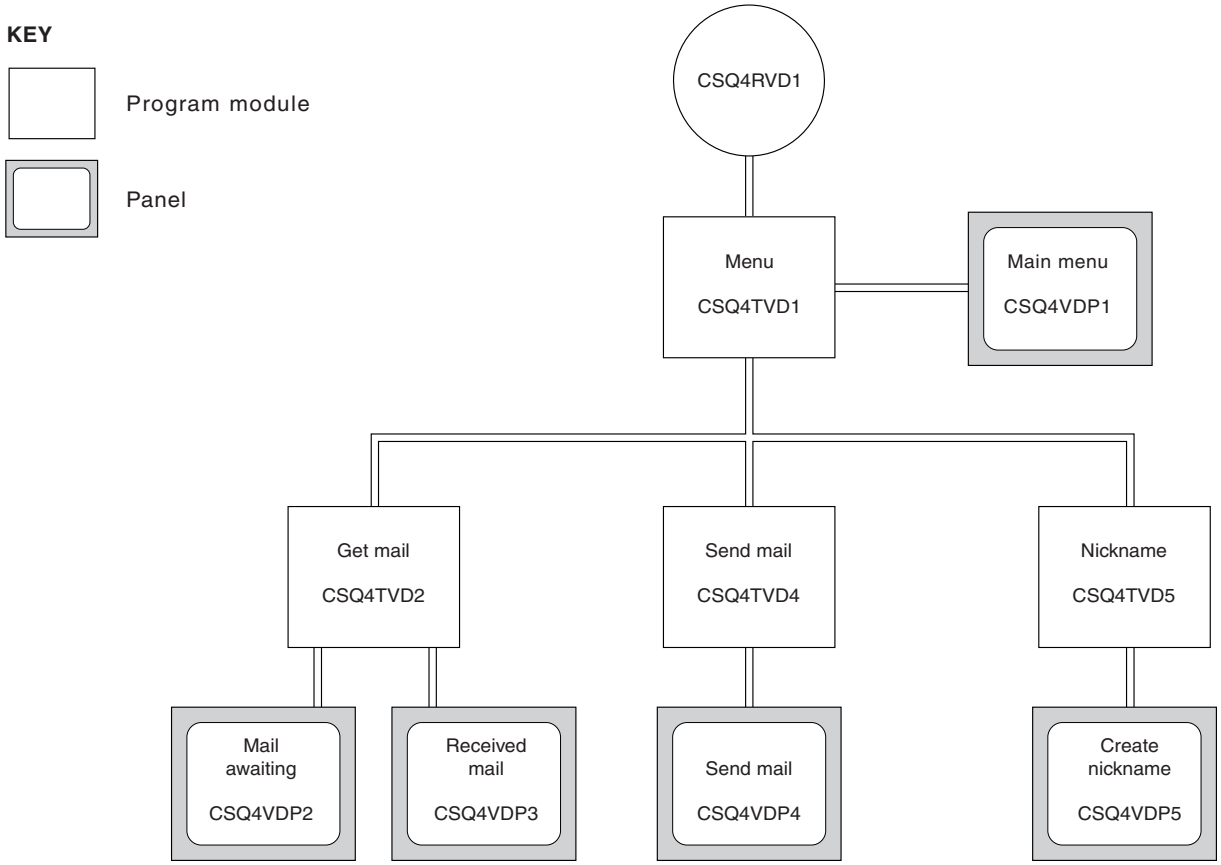


Figure 49. Programs and panels for the TSO versions of the Mail Manager. This figure shows the names for the COBOL version.

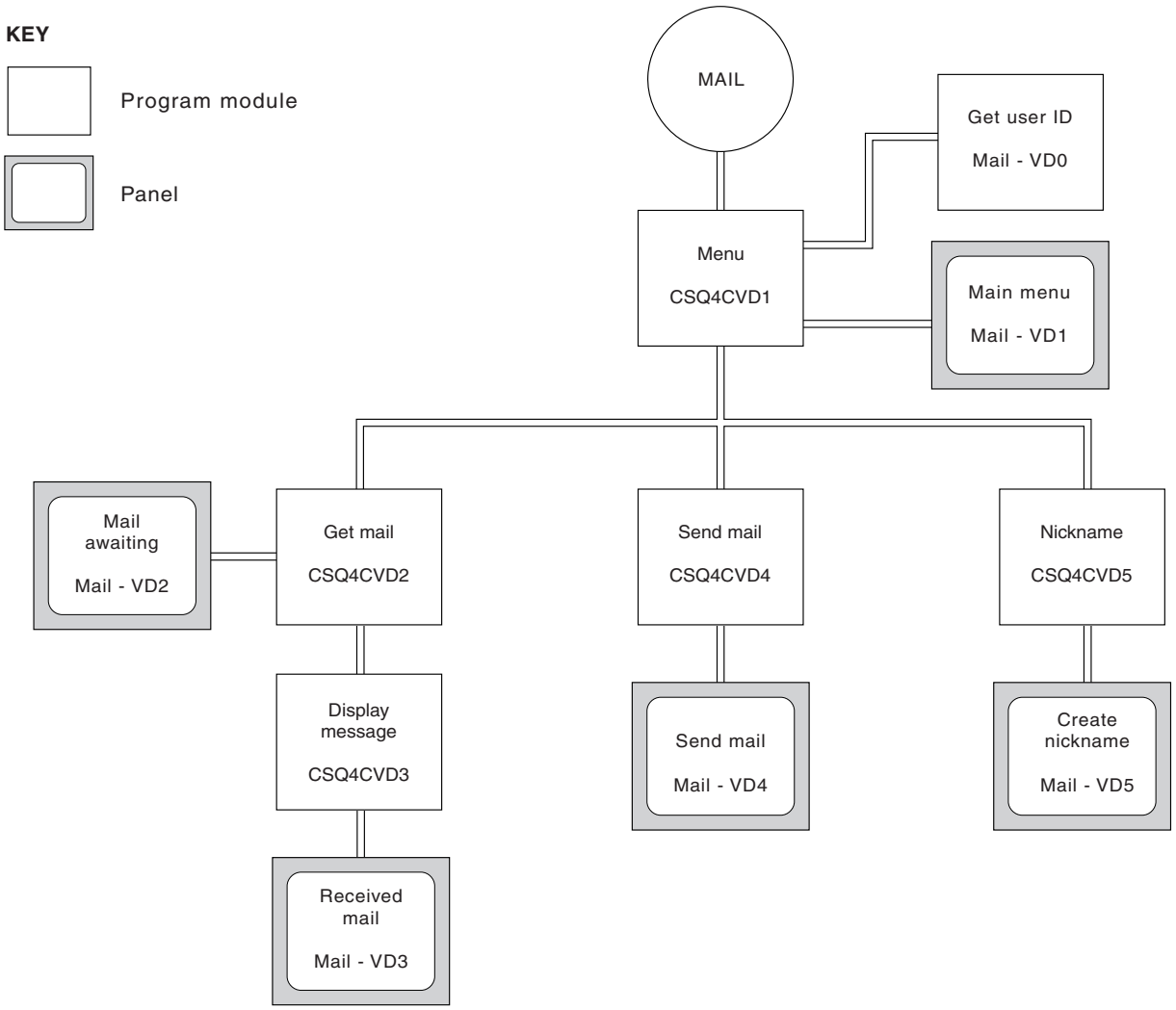


Figure 50. Programs and panels for the CICS version of the Mail Manager

Menu program:

In the TSO environment, the menu program is invoked by the CLIST. In the CICS environment, the program is invoked by transaction MAIL.

The menu program (CSQ4TVD1 for TSO, CSQ4CVD1 for CICS) is the initial program in the suite. It displays the menu (CSQ4VDP1 for TSO, VD1 for CICS) and invokes the other programs when they are selected from the menu.

The program first obtains the user's ID:

- In the CICS version of the program, if the user has signed on to CICS, the user ID is obtained by using the CICS command ASSIGN USERID. If the user has not signed on, the program displays the signon panel (CSQ4VD0) to prompt the user to enter a user ID. There is no security processing within this program; the user can give ***any*** user ID.
- In the TSO version, the user's ID is obtained from TSO in the CLIST. It is passed to the menu program as a variable in the ISPF shared pool.

After the program has obtained the user ID, it checks to ensure that the user has a mail queue (CSQ4SAMP.MAILMGR.*userid*). If a mail queue does not exist, the

program creates one by putting a message on the system-command input queue. The message contains the WebSphere MQ for z/OS command DEFINE QLOCAL. The object definition that this command uses sets the maximum depth of the queue to 9999 messages.

The program also creates a temporary dynamic queue to handle replies from the system-command input queue. To do this, the program uses the MQOPEN call, specifying the SYSTEM.DEFAULT.MODEL.QUEUE as the template for the dynamic queue. The queue manager creates the temporary dynamic queue with a name that has the prefix CSQ4SAMP; the remainder of the name is generated by the queue manager.

The program then opens the user's mail queue and finds the number of messages on the queue by inquiring about the current depth of the queue. To do this, the program uses the MQINQ call, specifying the MQIA_CURRENT_Q_DEPTH selector.

The program then performs a loop that displays the menu and processes the selection that the user makes. The loop is stopped when the user presses the PF3 key. When a valid selection is made, the appropriate program is started; otherwise an error message is displayed.

Get-mail and display-message programs:

In the TSO versions of the application, the get-mail and display-message functions are performed by the same program (CSQ4TVD2). In the CICS version of the application, these functions are performed by separate programs (CSQ4CVD2 and CSQ4CVD3).

The Mail Awaiting panel (CSQ4VDP2 for TSO, VD2 for CICS; see Figure 51 for an example) shows all the messages that are on the user's mail queue. To create this list, the program uses the MQGET call to browse all the messages on the queue, saving information about each one. In addition to the information displayed, the program records the *MsgId* and *CorrelId* of each message.

```
---------------------- WebSphere MQ for z/OS Sample Programs ------- ROW 16 OF 29
COMMAND ==>                                              Scroll    ===> PAGE
                                                           USERID - NTSFV02
                              Mail Manager System          QMGR   - VC4
                                 Mail Awaiting

                       Msg      Mail        Date       Time
                       No       From        Sent       Sent
                  16
                       16       Deleted
                       17       JOHNJ       01/06/1993  12:52:02
                       18       JOHNJ       01/06/1993  12:52:02
                       19       JOHNJ       01/06/1993  12:52:03
                       20       JOHNJ       01/06/1993  12:52:03
                       21       JOHNJ       01/06/1993  12:52:03
                       22       JOHNJ       01/06/1993  12:52:04
                       23       JOHNJ       01/06/1993  12:52:04
                       24       JOHNJ       01/06/1993  12:52:04
                       25       JOHNJ       01/06/1993  12:52:05
                       26       JOHNJ       01/06/1993  12:52:05
                       27       JOHNJ       01/06/1993  12:52:05
                       28       JOHNJ       01/06/1993  12:52:06
                       29       JOHNJ       01/06/1993  12:52:06
```

Figure 51. Example of a panel showing a list of waiting messages

From the Mail Awaiting panel the user can select one message and display the contents of the message (see Figure 52 for an example). The program uses the MQGET call to remove this message from the queue, using the *MsgId* and *CorrelId* that the program noted when it browsed all the messages. This MQGET call is performed using the MQGMO_SYNCPOINT option. The program displays the contents of the message, then declares a syncpoint: this commits the MQGET call, so the message now no longer exists.

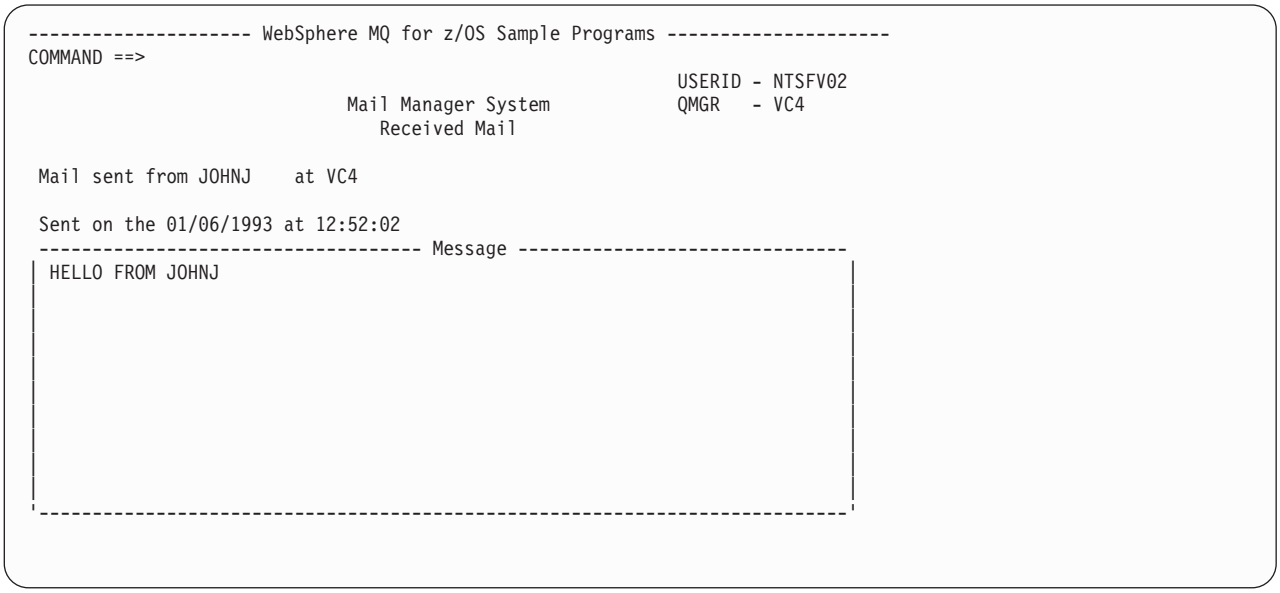


Figure 52. Example of a panel showing the contents of a message

An obvious extension to the function provided by the Mail Manager is to give the user the option to leave the message on the queue after viewing its contents. To do this, you would have to back out the MQGET call that removes the message from the queue, after displaying the message.

Send-mail program:

When the user has completed the Send Mail panel (CSQ4VDP4 for TSO, VD4 for CICS), the send-mail program (CSQ4TVD4 for TSO, CSQ4CVD4 for CICS) puts the message on the receiver's mail queue.

To do this, the program uses the MQPUT1 call. The destination of the message depends on how the user has filled the fields in the Send Mail panel:

- If the user has specified only a user name, the program first assumes that the name is a nickname, and sends the message to the object defined by that name. If there is no such nickname, the program attempts to send the message to a local queue of that name.
- If the user has specified both a user name and a queue manager name, the program sends the message to the mail queue that is defined by those two names.

The program does not accept blank messages, and it removes leading blanks from each line of the message text.

If the MQPUT1 call is successful, the program displays a message that shows that the user name and queue manager name to which the message was put. If the call is unsuccessful, the program checks specifically for the reason codes that indicate

the queue or the queue manager do not exist; these are MQRC_UNKNOWN_OBJECT_NAME and MQRC_UNKNOWN_OBJECT_Q_MGR. The program displays its own error message for each of these errors; for other errors, the program displays the completion and reason codes returned by the call.

Nickname program:

When the user defines a nickname, the nickname program (CSQ4TVD5 for TSO, CSQ4CVD5 for CICS) creates a queue that has the nickname as part of its name.

The program does this by putting a message on the system-command input queue. The message contains the WebSphere MQ for z/OS command DEFINE QALIAS or DEFINE QREMOTE. The type of queue that the program creates depends on how the user has filled the fields of the Create Nickname panel (CSQ4VDP5 for TSO, VD5 for CICS):

- If the user has specified only a user name, or the queue manager name is the same as that of the queue manager to which the Mail Manager is connected, the program creates an alias queue.
- If the user has specified both a user name and a queue manager name (and the queue manager is not the one to which the Mail Manager is connected), the program creates a local definition of a remote queue. The program does not check the existence of the queue to which this definition resolves, or even that the remote queue manager exists.

The program also creates a temporary dynamic queue to handle replies from the system-command input queue.

If the queue manager cannot create the nickname queue for a reason that the program expects (for example, the queue already exists), the program displays its own error message. If the queue manager cannot create the queue for a reason that the program does not expect, the program displays up to two of the error messages that are returned to the program by the command server.

Note: For each nickname, the nickname program creates only an alias queue or a local definition of a remote queue. The local queues to which these queue names resolve are created only when the user ID that is contained in the nickname is used to start the Mail Manager application.

The Credit Check sample

The Credit Check sample application is a suite of programs that demonstrates how to use many of the features provided by WebSphere MQ for z/OS. It shows how the many component programs of an application can pass messages to each other using message queuing techniques.

The sample can run as a standalone CICS application. However, to demonstrate how to design a message queuing application that uses the facilities provided by both the CICS and IMS environments, one module is also supplied as an IMS batch message processing program. This extension to the sample is described in “The IMS extension to the Credit Check sample” on page 498.

You can also run the sample on more than one queue manager, and send messages between each instance of the application. To do this, see “The Credit Check sample with multiple queue managers” on page 498.

The CICS programs are delivered in C and COBOL. The single IMS program is delivered only in C. The supplied data sets are shown in Table 42 on page 463 and Table 43 on page 465.

The application demonstrates a method of assessing the risk when bank customers ask for loans. The application shows how a bank could work in two ways to process loan requests:

- When dealing directly with a customer, bank staff want immediate access to account and credit-risk information.
- When dealing with written applications, bank staff can submit a series of requests for account and credit-risk information, and deal with the replies at a later time.

The financial and security details in the application have been kept simple so that the message queuing techniques are clear.

Preparing and running the Credit Check sample

To prepare and run the Credit Check sample, perform the following steps:

1. Create the VSAM data set that holds information about some example accounts. Do this by editing and running the JCL supplied in data set CSQ4FILE.
2. Perform the steps in "Preparing the sample applications for the CICS environment" on page 461. (The additional steps that you must perform if you want to use the IMS extension to the sample are described in "The IMS extension to the Credit Check sample" on page 498.)
3. Start the CKTI trigger monitor (supplied with WebSphere MQ for z/OS) against queue CSQ4SAMP.INITIATION.QUEUE, using the CICS transaction CKQC.
4. To start the application, start your CICS system and use the transaction `MVB1`.
5. Select **Immediate** or **Batch** inquiry from the first panel.

   The immediate and batch inquiry panels are similar; Figure 53 on page 487 shows the Immediate Inquiry panel.

```
CSQ4VB2             WebSphere MQ for z/OS Sample Programs

                      Credit Check - Immediate Inquiry

Specify details of the request, then press Enter.
      Name . . . . . .  _____________________
      Social security number  ___ __ ____
      Bank account name  . .  ______________________________
      Account number . . . .  _________
      Amount requested . . .  012345
Response from CHECKING ACCOUNT for name : _____________________
      Account information not found
      Credit worthiness index - NOT KNOWN
..
..
..
..
..
..
..
..
..
MESSAGE LINE
F1=Help  F3=Exit  F5=Make another inquiry
```

Figure 53. Immediate Inquiry panel for the Credit Check sample application

6. Enter an account number and loan amount in the appropriate fields. See "Entering information in the inquiry panels" for guidance on what information to enter in these fields.

Entering information in the inquiry panels:

The Credit Check sample application checks that the data you enter in the **Amount requested** field of the inquiry panels is in the form of integers.

If you enter one of the following account numbers, the application finds the appropriate account name, average account balance, and credit worthiness index in the VSAM data set CSQ4BAQ:

- 2222222222
- 3111234329
- 3256478962
- 3333333333
- 3501676212
- 3696879656
- 4444444444
- 5555555555
- 6666666666
- 7777777777

You can enter any, or no, information in the other fields. The application retains any information that you enter and returns the same information in the reports that it generates.

Design of the sample

This section describes the design of each of the programs that comprise the Credit Check sample application.

For a discussion of some of the techniques that were considered during the design of the application, see “Design considerations” on page 495.

Figure 54 on page 489 shows the programs that make up the application, and also the queues that these programs serve. In this figure, the prefix CSQ4SAMP has been omitted from all the queue names to make the figure easier to understand.

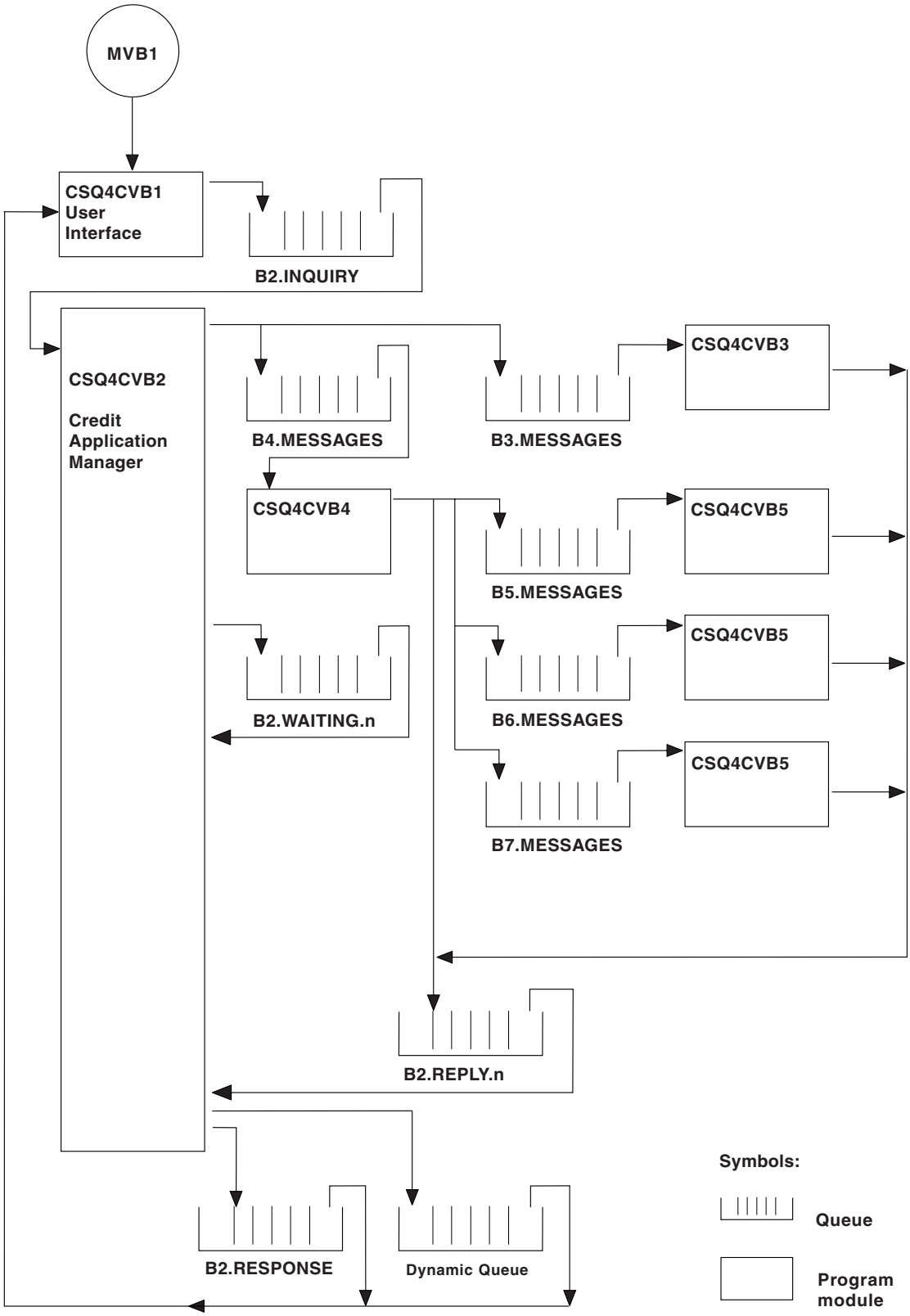


Figure 54. Programs and queues for the Credit Check sample application (COBOL programs only). In the sample application, the queue names shown in this figure have the prefix CSQ4SAMP.

User interface program (CSQ4CVB1):

When you start the conversational-mode CICS transaction MVB1, this starts the user interface program for the application.

This program puts inquiry messages on queue CSQ4SAMP.B2.INQUIRY and gets replies to those inquiries from a reply-to queue that it specifies when it makes the inquiry. From the user interface you can submit either immediate or batch inquiries:

- For immediate inquiries, the program creates a temporary dynamic queue that it uses as a reply-to queue. This means that each inquiry has its own reply-to queue.
- For batch inquiries, the user-interface program gets replies from the queue CSQ4SAMP.B2.RESPONSE. For simplicity, the program gets replies for all its inquiries from this one reply-to queue. It is easy to see that a bank might want to use a separate reply-to queue for each user of MVB1, so that they could each see replies to only those inquiries that they had initiated.

Important differences between the properties of messages used in the application when in batch and immediate mode are:

- For batch working, the messages have a low priority, so they are processed after any loan requests that are entered in immediate mode. Also, the messages are persistent, so they are recovered if the application or the queue manager has to restart.
- For immediate working, the messages have a high priority, so they are processed before any loan requests that are entered in batch mode. Also, messages are not persistent so they are discarded if the application or the queue manager has to restart.

However, in all cases, the properties of loan request messages are propagated throughout the application. So, for example, all messages that result from a high-priority request will also have a high priority.

Credit application manager (CSQ4CVB2):

The Credit Application Manager (CAM) program performs most of the processing for the Credit Check application.

The CAM is started by the CKTI trigger monitor (supplied with WebSphere MQ for z/OS) when a trigger event occurs on either queue CSQ4SAMP.B2.INQUIRY or queue CSQ4SAMP.B2.REPLY.*n*, where *n* is an integer that identifies one of a set of reply queues. The trigger message contains data that includes the name of the queue on which the trigger event occurred.

The CAM uses queues with names of the form CSQ4SAMP.B2.WAITING.n to store information about inquiries that it is processing. The queues are named so that they are each paired with a reply-to queue; for example, queue CSQ4SAMP.B2.WAITING.3 contains the input data for a particular inquiry, and queue CSQ4SAMP.B2.REPLY.3 contains a set of reply messages (from programs that query databases) all relating to that same inquiry. To understand the reasons behind this design, see “Separate inquiry and reply queues in the CAM” on page 495.

Startup logic:

If the trigger event occurs on queue CSQ4SAMP.B2.INQUIRY, the CAM opens the queue for shared access. It then tries to open each reply queue until a free one is found. If it cannot find a free reply queue, the CAM logs the fact and terminates normally.

If the trigger event occurs on queue CSQ4SAMP.B2.REPLY.n, the CAM opens the queue for exclusive access. If the return code reports that the object is already in use, the CAM terminates normally. If any other error occurs, the CAM logs the error and terminates. The CAM opens the corresponding waiting queue and the inquiry queue, then starts getting and processing messages. From the waiting queue, the CAM recovers details of partially-completed inquiries.

For the sake of simplicity in this sample, the names of the queues used are held in the program. In a business environment, the queue names would probably be held in a file accessed by the program.

Getting a message:

The CAM first attempts to get a message from the inquiry queue using the MQGET call with the MQGMO_SET_SIGNAL option. If a message is available immediately, the message is processed; if no message is available, a signal is set.

The CAM then attempts to get a message from the reply queue, again using the MQGET call with the same option. If a message is available immediately, the message is processed; otherwise a signal is set.

When both signals are set, the program waits until one of the signals is posted. If a signal is posted to indicate that a message is available, the message is retrieved and processed. If the signal expires or the queue manager is terminating, the program terminates.

Processing the message retrieved:

A message retrieved by the CAM can be one of four types:

- An inquiry message
- A reply message
- A propagation message
- An unexpected or unwanted message

The CAM processes these messages as follows:

Inquiry message

Inquiry messages come from the user interface program. It creates an inquiry message for each loan request.

For all loan requests, the CAM requests the average balance of the customer's checking account. It does this by putting a request message on alias queue CSQ4SAMP.B2.OUTPUT.ALIAS. This queue name resolves to queue CSQ4SAMP.B3.MESSAGES, which is processed by the checking-account program, CSQ4CVB3. When the CAM puts a message on this alias queue, it specifies the appropriate CSQ4SAMP.B2.REPLY.n queue for the reply-to queue. An alias queue is used here so that program CSQ4CVB3 can easily be replaced by another program that processes a base queue of a different name. To do this, you simply redefine the alias queue so that its name resolves to the new queue. Also, you could assign differing access authorities to the alias queue and to the base queue.

If a user requests a loan that is larger than 10000 units, the CAM initiates checks on other databases as well. It does this by putting a request message on queue CSQ4SAMP.B4.MESSAGES, which is processed by the distribution program, CSQ4CVB4. The process serving this queue propagates the message to queues served by programs that have access to other records such as credit card history, savings accounts, and mortgage payments. The data from these programs is returned to the reply-to queue specified in the put operation. Additionally, a propagation message is sent to the reply-to queue by this program to specify how many propagation messages have been sent.

In a business environment, the distribution program would probably reformat the data provided to match the format required by each of the other types of bank account.

Any of the queues referred to here can be on a remote system.

For each inquiry message, the CAM initiates an entry in the memory-resident Inquiry Record Table (IRT). This record contains:

- The *MsgId* of the inquiry message
- In the ReplyExp field, the number of responses expected (equal to the number of messages sent)
- In the ReplyRec field, the number of replies received (zero at this stage)
- In the PropsOut field, an indication of whether a propagation message is expected

The CAM copies the inquiry message onto the waiting queue with:

- *Priority* set to 3
- *CorrelId* set to the *MsgId* of the inquiry message
- The other message-descriptor fields set to those of the inquiry message

Propagation message

A propagation message contains the number of queues to which the distribution program has forwarded the inquiry. The message is processed as follows:

1. Add to the ReplyExp field of the appropriate record in the IRT the number of messages sent. This information is in the message.
2. Increment by 1 the ReplyRec field of the record in the IRT.
3. Decrement by 1 the PropsOut field of the record in the IRT.
4. Copy the message onto the waiting queue. The CAM sets the *Priority* to 2 and the other fields of the message descriptor to those of the propagation message.

Reply message

A reply message contains the response to one of the requests to the checking-account program or to one of the agency-query programs. Reply messages are processed as follows:

1. Increment by 1 the ReplyRec field of the record in the IRT.
2. Copy the message onto the waiting queue with *Priority* set to 1 and the other fields of the message descriptor set to those of the reply message.
3. If ReplyRec = ReplyExp, and PropsOut = 0, set the MsgComplete flag.

Other messages

The application does not expect other messages. However, the application might receive messages broadcast by the system, or reply messages with unknown *CorrelId*s.

The CAM puts these messages on queue CSQ4SAMP.DEAD.QUEUE, where they can be examined. If this put operation fails, the message is lost and the program continues. For more information on the design of this part of the program, see "How the sample handles unexpected messages" on page 496.

Sending an answer:

When the CAM has received all the replies it is expecting for an inquiry, it processes the replies and creates a single response message.

It consolidates into one message all the data from all reply messages that have the same *CorrelId*. This response is put on the reply-to queue specified in the original loan request. The response message is put within the same unit of work that contains the retrieval of the final reply message. This is to simplify recovery by ensuring that there is never a completed message on queue CSQ4SAMP.B2.WAITING.n.

Recovery of partially-completed inquiries:

The CAM copies onto queue CSQ4SAMP.B2.WAITING.n all the messages that it receives. It sets the fields of the message descriptor like this:

- *Priority* is determined by the type of message:
  - For request messages, priority = 3
  - For datagrams, priority = 2
  - For reply messages, priority = 1
- *CorrelId* is set to the *MsgId* of the loan request message
- Other MQMD fields are copied from those of the received message

When an inquiry has been completed, the messages for a specific inquiry are removed from the waiting queue during answer processing. Therefore, at any time, the waiting queue contains all messages relevant to in-progress inquiries. These messages are used to recover details of in-progress inquiries if the program has to restart. The different priorities are set so that inquiry messages are recovered before propagations or reply messages.

Checking-account program (CSQ4CVB3):

The checking-account program is started by a trigger event on queue CSQ4SAMP.B3.MESSAGES. After it has opened the queue, this program gets a message from the queue using the MQGET call with the wait option, and with the wait interval set to 30 seconds.

The program searches VSAM data set CSQ4BAQ for the account number in the loan request message. It retrieves the corresponding account name, average balance, and credit worthiness index, or notes that the account number is not in the data set.

The program then puts a reply message (using the MQPUT1 call) on the reply-to queue named in the loan request message. For this reply message, the program:

- Copies the *CorrelId* of the loan request message
- Uses the MQPMO_PASS_IDENTITY_CONTEXT option

The program continues to get messages from the queue until the wait interval expires.

Distribution program (CSQ4CVB4):

The distribution program is started by a trigger event on queue CSQ4SAMP.B4.MESSAGES. To simulate the distribution of the loan request to other agencies that have access to records such as credit card history, savings accounts, and mortgage payments, the program puts a copy of the same message on all the queues in the namelist CSQ4SAMP.B4.NAMELIST. There are three of these queues, with names of the form CSQ4SAMP.B*n*.MESSAGES, where *n* is 5, 6, or 7. In a business application, the agencies could be at separate locations, so these queues could be remote queues. If you want to modify the sample application to show this, see "The Credit Check sample with multiple queue managers" on page 498.

The distribution program performs the following steps:

1. From the namelist, gets the names of the queues that the program is to use. The program does this by using the MQINQ call to inquire about the attributes of the namelist object.
2. Opens these queues and also CSQ4SAMP.B4.MESSAGES.
3. Performs the following loop until there are no more messages on queue CSQ4SAMP.B4.MESSAGES:
   a. Get a message using the MQGET call with the wait option, and with the wait interval set to 30 seconds.
   b. Put a message on each queue listed in the namelist, specifying the name of the appropriate CSQ4SAMP.B2.REPLY.n queue for the reply-to queue. The program copies the *CorrelId* of the loan request message to these copy messages, and it uses the MQPMO_PASS_IDENTITY_CONTEXT option on the MQPUT call.
   c. Send a datagram message to queue CSQ4SAMP.B2.REPLY.n to show how many messages it has successfully put.
   d. Declare a syncpoint.

Agency-query program (CSQ4CVB5/CSQ4CCB5):

The agency-query program is supplied as both a COBOL program and a C program. Both programs have the same design. This shows that programs of different types can easily coexist within a WebSphere MQ application, and that the program modules that comprise such an application can easily be replaced.

An instance of the program is started by a trigger event on any of these queues:

- For the COBOL program (CSQ4CVB5):
  - CSQ4SAMP.B5.MESSAGES
  - CSQ4SAMP.B6.MESSAGES
  - CSQ4SAMP.B7.MESSAGES
- For the C program (CSQ4CCB5), queue CSQ4SAMP.B8.MESSAGES

Note: If you want to use the C program, you must alter the definition of the namelist CSQ4SAMP.B4.NAMELIST to replace the queue CSQ4SAMP.B7.MESSAGES with CSQ4SAMP.B8.MESSAGES. To do this, you can use any one of:

- The WebSphere MQ for z/OS operations and control panels
- The ALTER NAMELIST command (described in the WebSphere MQ Script (MQSC) Command Reference)
- The CSQUTIL utility (described in the WebSphere MQ for z/OS System Administration Guide)

After it has opened the appropriate queue, this program gets a message from the queue using the MQGET call with the wait option, and with the wait interval set to 30 seconds.

The program simulates the search of an agency's database by searching the VSAM data set CSQ4BAQ for the account number that was passed in the loan request message. It then builds a reply that includes the name of the queue that it is serving and a creditworthiness index. To simplify the processing, the creditworthiness index is selected at random.

When putting the reply message, the program uses the MQPUT1 call and:

- Copies the *CorrelId* of the loan request message
- Uses the MQPMO_PASS_IDENTITY_CONTEXT option

The program sends the reply message to the reply-to queue named in the loan request message. (The name of the queue manager that owns the reply-to queue is also specified in the loan request message.)

Design considerations

This section discusses:

- Why the CAM uses separate inquiry and reply queues
- How the sample handles errors
- How the sample handles unexpected messages
- How the sample uses syncpoints
- How the sample uses message context information

Separate inquiry and reply queues in the CAM:

The application could use a single queue for both inquiries and replies, but it was designed to use separate queues for the following reasons:

- When the program is handling the maximum number of inquiries, further inquiries can be left on the queue. If a single queue were being used, these would have to be taken off the queue and stored elsewhere.
- Other instances of the CAM could be started automatically to service the same inquiry queue if message traffic was high enough to warrant it. But the program must track in-progress inquiries, and to do this it must get back all replies to inquiries it has initiated. If only one queue were used, the program would have to browse the messages to see if they were for this program or for another. This would make the operation much less efficient.

  The application can support multiple CAMs and can recover in-progress inquiries effectively by using paired reply-to and waiting queues.
- The program can wait on multiple queues effectively by using signaling.

How the sample handles errors:

The user interface program handles errors simply by reporting them directly to the user.

The other programs do not have user interfaces, so they have to handle errors in other ways. Also, in many situations (for example, if an MQGET call fails) these other programs do not know the identity of the user of the application.

The other programs put error messages on a CICS temporary storage queue called CSQ4SAMP. You can browse this queue using the CICS-supplied transaction CEBR. The programs also write error messages to the CICS CSML log.

How the sample handles unexpected messages:

When you design a message-queuing application, you must decide how to handle messages that arrive on a queue unexpectedly.

The two basic choices are:

- The application does no more work until it has processed the unexpected message. This probably means that the application notifies an operator, terminates itself, and ensures that it is not restarted automatically (it can do this by setting triggering off). This choice means that all processing for the application can be halted by a single unexpected message, and the intervention of an operator is required to restart the application.
- The application removes the message from the queue it is serving, puts the message in another location, and continues processing. The best place to put this message is on the system dead-letter queue.

If you choose the second option:

- An operator, or another program, should examine the messages that are put on the dead-letter queue to find out where the messages are coming from.
- An unexpected message is lost if it cannot be put on the dead-letter queue.
- An long unexpected message is truncated if it is longer than the limit for messages on the dead-letter queue, or longer than the buffer size in the program.

To ensure that the application smoothly handles all inquiries with minimal impact from outside activities, the Credit Check sample application uses the second option. To allow you to keep the sample separate from other applications that use the same queue manager, the Credit Check sample does not use the system dead-letter queue; instead, it uses its own dead-letter queue. This queue is named CSQ4SAMP.DEAD.QUEUE. The sample truncates any messages that are longer than the buffer area provided for the sample programs. You can use the Browse sample application to browse messages on this queue, or use the Print Message sample application to print the messages together with their message descriptors.

However, if you extend the sample to run across more than one queue manager, unexpected messages, or messages that cannot be delivered, could be put on the system dead-letter queue by the queue manager.

How the sample uses syncpoints:

The programs in the Credit Check sample application declare syncpoints to ensure that:

- Only one reply message is sent in response to each expected message

- Multiple copies of unexpected messages are never put on the sample's dead-letter queue
- The CAM can recover the state of all partially-completed inquiries by getting persistent messages from its waiting queue

To achieve this, a single unit of work is used to cover the getting of a message, the processing of that message, and any subsequent put operations.

How the sample uses message context information:

When the user interface program (CSQ4CVB1) sends messages, it uses the MQPMO_DEFAULT_CONTEXT option. This means that the queue manager generates both identity and origin context information. The queue manager gets this information from the transaction that started the program (MVB1) and from the user ID that started the transaction.

When the CAM sends inquiry messages, it uses the MQPMO_PASS_IDENTITY_CONTEXT option. This means that the identity context information of the message being put is copied from the identity context of the original inquiry message. With this option, origin context information is generated by the queue manager.

When the CAM sends reply messages, it uses the MQPMO_ALTERNATE_USER_AUTHORITY option. This causes the queue manager to use an alternate user ID for its security check when the CAM opens a reply-to queue. The CAM uses the user ID of the submitter of the original inquiry message. This means that users are allowed to see replies to only those inquiries that they have originated. The alternate user ID is obtained from the identity context information in the message descriptor of the original inquiry message.

When the query programs (CSQ4CVB3/4/5) send reply messages, they use the MQPMO_PASS_IDENTITY_CONTEXT option. This means that the identity context information of the message being put is copied from the identity context of the original inquiry message. With this option, origin context information is generated by the queue manager.

Note: The user ID associated with the MVB3/4/5 transactions requires access to the B2.REPLY.n queues. These user IDs might not be the same as those associated with the request being processed. To get around this possible security exposure, the query programs could use the MQPMO_ALTERNATE_USER_AUTHORITY option when putting their replies. This would mean that each individual user of MVB1 needs authority to open the B2.REPLY.n queues.

Use of message and correlation identifiers in the CAM:

The application has to monitor the progress of all the *live* inquiries it is processing at any one time. To do this it uses the unique message identifier of each loan request message to associate all the information that it has about each inquiry.

The CAM copies the *MsgId* of the inquiry message into the *CorrelId* of all the request messages it sends for that inquiry. The other programs in the sample (CSQ4CVB3 - 5) copy the *CorrelId* of each message that they receive into the *CorrelId* of their reply message.

The Credit Check sample with multiple queue managers

You can use the Credit Check sample application to demonstrate distributed queuing by installing the sample on two queue managers and CICS systems (with each queue manager connected to a different CICS system).

When the sample program is installed, and the trigger monitor (CKTI) is running on each system, you need to:

1. Set up the communication link between the two queue managers. For information on how to do this, see WebSphere MQ Intercommunication.
2. On one queue manager, create a local definition for each of the remote queues (on the other queue manager) that you want to use. These queues can be any of CSQ4SAMP.B*n*.MESSAGES, where *n* is 3, 5, 6, or 7. (These are the queues that are served by the checking-account program and the agency-query program.) For information on how to do this, see the WebSphere MQ Script (MQSC) Command Reference.
3. Change the definition of the namelist (CSQ4SAMP.B4.NAMELIST) so that it contains the names of the remote queues that you want to use. For information on how to do this, see the WebSphere MQ Script (MQSC) Command Reference.

The IMS extension to the Credit Check sample

A version of the checking-account program is supplied as an IMS batch message processing (BMP) program. It is written in the C language.

The program performs the same function as the CICS version, except that to obtain the account information, the program reads an IMS database instead of a VSAM file. If you replace the CICS version of the checking-account program with the IMS version, you see no difference in the method of using the application.

To prepare and run the IMS version you must:

1. Follow the steps in "Preparing and running the Credit Check sample" on page 486.
2. Follow the steps in "Preparing the sample application for the IMS environment" on page 464.
3. Alter the definition of the alias queue CSQ4SAMP.B2.OUTPUT.ALIAS to resolve to queue CSQ4SAMP.B3.IMS.MESSAGES (instead of CSQ4SAMP.B3.MESSAGES). To do this, you can use one of:
   - The WebSphere MQ for z/OS operations and control panels
   - The ALTER QALIAS command (described in the WebSphere MQ Script (MQSC) Command Reference)

Another way of using the IMS checking-account program is to make it serve one of the queues that receives messages from the distribution program. In the delivered form of the Credit Check sample application, there are three of these queues (B5/6/7.MESSAGES), all served by the agency-query program. This program searches a VSAM data set. To compare the use of the VSAM data set and the IMS database, you could make the IMS checking-account program serve one of these queues instead. To do this, you must alter the definition of the namelist CSQ4SAMP.B4.NAMELIST to replace one of the CSQ4SAMP.B*n*.MESSAGES queues with the CSQ4SAMP.B3.IMS.MESSAGES queue. You can use one of:

- The WebSphere MQ for z/OS operations and control panels
- The ALTER NAMELIST command (described in the WebSphere MQ Script (MQSC) Command Reference)

You can then run the sample from CICS transaction MVB1 as usual. The user sees no difference in operation or response. The IMS BMP stops either after receiving a stop message or after being inactive for five minutes.

Design of the IMS checking-account program (CSQ4ICB3):

This program runs as a BMP. Start the program using its JCL before any WebSphere MQ messages are sent to it.

The program searches an IMS database for the account number in the loan request messages. It retrieves the corresponding account name, average balance, and credit worthiness index.

The program sends the results of the database search to the reply-to queue named in the WebSphere MQ message being processed. The message returned appends the account type and the results of the search to the message received so that the transaction building the response can confirm that the correct query is being processed. The message is in the form of three 79-character groups, as follows:

```
'Response from CHECKING ACCOUNT for name : JONES J B'
'      Opened 870530, 3-month average balance = 000012.57'
'      Credit worthiness index - BBB'
```

When running as a message-oriented BMP, the program drains the IMS message queue, then reads messages from the WebSphere MQ for z/OS queue and processes them. No information is received from the IMS message queue. The program reconnects to the queue manager after each checkpoint because the handles have been closed.

When running in a batch-oriented BMP, the program continues to be connected to the queue manager after each checkpoint because the handles are not closed.

The Message Handler sample

The Message Handler sample TSO application allows you to browse, forward, and delete messages on a queue. The sample is available in C and COBOL.

Preparing and running the sample

Follow these steps:

1. Prepare the sample as described in “Preparing sample applications for the TSO environment” on page 459.
2. Tailor the CLIST (CSQ4RCH1) provided for the sample to define the location of the panels, the location of the message file, and the location of the load modules.

You can use CLIST CSQ4RCH1 to run both the C and the COBOL version of the sample. The supplied version of CSQ4RCH1 runs the C version, and contains instructions on the tailoring necessary for the COBOL version.

Note:

1. There are no sample queue definitions provided with the sample.
2. VS COBOL II does not support multitasking with ISPF, so do not use the Message Handler sample application on both sides of a split screen. If you do, the results are unpredictable.

Using the sample

Having installed the sample and invoked it from the tailored CLIST CSQ4RCH1, the screen shown in Figure 55 is displayed.

```
---------------------- WebSphere MQ for z/OS -- Samples -----------------------
 COMMAND ===>
                                                          User Id : JOHNJ



  Enter information. Press ENTER :


   Queue Manager Name    : ________________________________________________ :

   Queue Name            : ________________________________________________ :









  F1=HELP      F2=SPLIT     F3=END       F4=RETURN    F5=RFIND     F6=RCHANGE
  F7=UP        F8=DOWN      F9=SWAP     F10=LEFT     F11=RIGHT    F12=RETRIEVE
```

Figure 55. Initial screen for Message Handler sample

Enter the queue manager and queue name to be viewed (case sensitive) and the message list screen is displayed (see Figure 56).

```
---------------------- WebSphere MQ for z/OS -- Samples ------- Row 1 to 4 of 4
COMMAND ==>

 Queue Manager    : VM03                                          :
 Queue            : MQEI.IMS.BRIDGE.QUEUE                         :

 Message number   01 of 04

 Msg   Put Date   Put Time  Format     User      Put Application
 No   MM/DD/YYYY  HH:MM:SS  Name   Identifier    Type     Name
 01   10/16/1998  13:51:19 MQIMS    NTSFV02      00000002 NTSFV02A
 02   10/16/1998  13:55:45 MQIMS    JOHNJ        00000011 EDIT\CLASSES\BIN\PROGTS
 03   10/16/1998  13:54:01 MQIMS    NTSFV02      00000002 NTSFV02B
 04   10/16/1998  13:57:22 MQIMS    johnj        00000011 EDIT\CLASSES\BIN\PROGTS
****************************** Bottom of data ********************************
```

Figure 56. Message list screen for Message Handler sample

This screen shows the first 99 messages on the queue and, for each, shows the following fields:

Msg No
Message number

Put Date MM/DD/YYYY
Date that the message was put on the queue (GMT)

Put Time HH:MM:SS
Time that the message was put on the queue (GMT)

Format Name
MQMD.Format field

User Identifier
MQMD.UserIdentifier field

Put Application Type
MQMD.PutApplType field

Put Application Name
MQMD.PutApplName field

The total number of messages on the queue is also displayed.

From this screen a message can be chosen, by number not by cursor position, and then displayed. For an example, see Figure 57.

```
---------------------- WebSphere MQ for z/OS -- Samples ----- Row 1 to 35 of 35
COMMAND ==>

 Queue Manager    : VM03                                              :
 Queue            : MQEI.IMS.BRIDGE.QUEUE                             :
 Forward to Q Mgr : VM03                                              :
 Forward to Queue : QL.TEST.ISCRES1                                   :

 Action : _ :   (D)elete  (F)orward

 Message Content :
------------------------------------------------------------------------------
 Message Descriptor
   StrucId          : `MD  `
   Version          : 000000001
   Report           : 000000000
   MsgType          : 000000001
   Expiry           : -00000001
   Feedback         : 000000000
   Encoding         : 000000785
   CodedCharSetId   : 000000500
   Format           : `MQIMS   `
   Priority         : 000000000
   Persistence      : 000000001
   MsgId            : `C3E2D840E5D4F0F34040404040404040AF6B30F0A89B7605`X
   CorrelId         : `00000000000000000000000000000000000000000000000000000000`X
   BackoutCount     : 000000000
   ReplyToQ         : `QL.TEST.ISCRES1                                    `
   ReplyToQMgr      : `VM03                                               `
   UserIdentifier   : `NTSFV02     `
   AccountingToken  :
          `06F2F5F5F3F0F1000000000000000000000000000000000000000000000000000000`X
   ApplIdentityData : `                                `
   PutApplType      : 000000002
   PutApplName      : `NTSFV02A                     `
   PutDate          : `19971016`
   PutTime          : `13511903`
   ApplOriginData   : `    `

 Message Buffer :   108 byte(s)
 00000000 :  C9C9 C840 0000 0001 0000 0054 0000 0311  `IIH .............`
 00000010 :  0000 0000 4040 4040 4040 4040 0000 0000  `....        ....`
 00000020 :  4040 4040 4040 4040 4040 4040 4040 4040  `                `
 00000030 :  4040 4040 4040 4040 4040 4040 4040 4040  `                `
 00000040 :  0000 0000 0000 0000 0000 0000 0000 0000  `................`
 00000050 :  40F1 C300 0018 0000 C9C1 D7D4 C4C9 F2F8  ` 1C.....IAPMDI28`
 00000060 :  40C8 C5D3 D3D6 40E6 D6D9 D3C4            ` HELLO WORLD    `
****************************** Bottom of data ********************************
```

Figure 57. Chosen message is displayed

Once the message has been displayed it can be deleted, left on the queue, or forwarded to another queue. The `Forward to Q Mgr` and `Forward to Queue` fields are initialized with values from the MQMD, these can be changed before forwarding the message.

The sample design allows only messages with unique `MsgId / CorrelId` combinations to be selected and displayed, because the message is retrieved using the `MsgId` and `CorrelId` as the key. If the key is not unique the sample cannot retrieve the chosen message with certainty.

Design of the sample

This section describes the design of each of the programs that comprise the Message Handler sample application.

Object validation program:

This requests a valid queue and queue manager name.

If you do not specify a queue manager name, the default queue manager is used, if available. Only local queues can be used; an MQINQ is issued to check the queue type and an error is reported if the queue is not local. If the queue is not opened successfully, or the MQGET call is inhibited on the queue, error messages are returned indicating the CompCode and Reason return code.

Message list program:

This displays a list of messages on a queue with information about them such as the putdate, puttime, and the message format.

The maximum number of messages stored in the list is 99. If there are more messages on the queue than this, the current queue depth is also displayed. To choose a message for display, type the message number into the entry field (the default is 01). If your entry is not valid, you receive an appropriate error message.

Message content program:

This displays message content.

The content is formatted and split into two parts:

1. Message descriptor
2. Message buffer

The message descriptor shows the contents of each field on a separate line.

The message buffer is formatted depending on its contents. If the buffer holds a dead letter header (MQDLH) or a transmission queue header (MQXQH), these are formatted and displayed before the buffer itself.

Before the buffer data is formatted, a title line shows the buffer length of the message in bytes. The maximum buffer size is 32768 bytes, and any message longer than this is truncated. The full size of the buffer is displayed along with a message indicating that only the first 32768 bytes of the message are displayed.

The buffer data is formatted in two ways:

1. After the offset into the buffer is printed, the buffer data is displayed in hexadecimal.
2. The buffer data is then displayed again as EBCDIC values. If any EBCDIC value cannot be printed, it prints a . (period) instead.

You can enter D for delete, or F for forward into the action field. If you choose to forward the message, the *forward-to queue* and *queue manager name* must be filled in appropriately. The defaults for these fields are read from the message descriptor ReplyToQ and ReplyToQMgr fields.

If you forward a message, any header block stored in the buffer is stripped. If the message is forwarded successfully, it is removed from the original queue. If you enter invalid actions, error messages are displayed.

An example help panel called CSQ4CHP9 is also available.

Chapter 5. C language examples

The extracts in this appendix are mostly taken from the WebSphere MQ for z/OS sample applications. They are applicable to all platforms, except where noted.

The examples in this appendix demonstrate the following techniques:

Connecting to a queue manager

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

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
#include <cmqc.h>
.
.
.
static char Parm1[MQ_Q_MGR_NAME_LENGTH] ;
.
.
.
int  main(int argc, char *argv[] )
   {
   /*                                                  */
   /*     Variables for MQ calls                       */
   /*                                                  */
   MQHCONN Hconn;      /* Connection handle            */
   MQLONG  CompCode;   /* Completion code              */
   MQLONG  Reason;     /* Qualifying reason            */
.
.
.
   /* Copy the queue manager name, passed in the       */
   /* parm field, to Parm1                             */
   strncpy(Parm1,argv[1],MQ_Q_MGR_NAME_LENGTH);
.
.
.
   /*                                                  */
   /* Connect to the specified queue manager.          */
   /*   Test the output of the connect call.  If the   */
   /*   call fails, print an error message showing the */
   /*   completion code and reason code, then leave the*/
   /*   program.                                       */
```

```
   /*                                                          */
   MQCONN(Parm1,
          &Hconn,
          &CompCode,
          &Reason);
   if ((CompCode != MQCC_OK) | (Reason != MQRC_NONE))
      {
      sprintf(pBuff, MESSAGE_4_E,
              ERROR_IN_MQCONN, CompCode, Reason);
      PrintLine(pBuff);
      RetCode = CSQ4_ERROR;
      goto AbnormalExit2;
      }
.
.
.
   }
```

Disconnecting from a queue manager

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

The variables used in this code extract are those that were set in "Connecting to a queue manager" on page 505. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

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

Creating a dynamic queue

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

This extract is taken from the Mail Manager sample application (program CSQ4TCD1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
.
.
.
MQLONG  HCONN = 0;   /* Connection handle      */
MQHOBJ  HOBJ;        /* MailQ Object handle    */
MQHOBJ  HobjTempQ;   /* TempQ Object Handle    */
MQLONG  CompCode;    /* Completion code        */
```

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

Opening an existing queue

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

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
#include <cmqc.h>
:
static char Parm1[MQ_Q_MGR_NAME_LENGTH];
:
int  main(int argc, char *argv[] )
   {
   /*
   /*     Variables for MQ calls                        */
   /*
   MQHCONN Hconn ;            /* Connection handle       */
```

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

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

Closing a queue

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

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
.
.
.
   /*                                                    */
   /*   Close the queue.                                 */
   /*   Test the output of the close call.  If the call  */
   /*   fails, print an error message showing the        */
```

```
   /*   completion code and reason code.                */
   /*                                                   */
   MQCLOSE(Hconn,
           &Hobj,
           MQCO_NONE,
           &CompCode,
           &Reason);
   if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
      {
      sprintf(pBuff, MESSAGE_4_E,
              ERROR_IN_MQCLOSE, CompCode, Reason);
      PrintLine(pBuff);
      RetCode = CSQ4_ERROR;
      }
.
.
.
```

Putting a message using MQPUT

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

This extract is not taken from the sample applications supplied with WebSphere MQ. For the names and locations of the sample applications, see "Sample programs (all platforms except z/OS)" on page 395 and "Sample programs for WebSphere MQ for z/OS" on page 452.

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

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

```
    /*-----------------------------------------------------*/
    MQPUT(Hconn, Hobj, &MsgDesc, &PutMsgOpts,
          sizeof(message_buffer), message_buffer,
          &CompCode, &Reason);
    /*-------------------------------------*/
    /*  Check completion and reason codes. */
    /*-------------------------------------*/
    switch (CompCode)
        {
         case MQCC_OK:
             break;
         case MQCC_FAILED:
             switch (Reason)
                 {
                  case MQRC_Q_FULL:
                  case MQRC_MSG_TOO_BIG_FOR_Q:
                      break;
                  default:
                      break; /* Perform error processing */
                  }
             break;
         default:
             break;          /* Perform error processing */
        }
}
```

Putting a message using MQPUT1

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

This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
.
.
.
MQLONG   Hconn;              /* Connection handle        */
MQHOBJ   Hobj_CheckQ;        /* Object handle            */
MQLONG   CompCode;           /* Completion code          */
MQLONG   Reason;             /* Qualifying reason        */
MQOD     ObjDesc    = {MQOD_DEFAULT};
                             /* Object descriptor        */
MQMD     MsgDesc    = {MQMD_DEFAULT};
                             /* Message descriptor       */
MQLONG   OpenOptions;        /* Control the MQOPEN call  */

MQGMO    GetMsgOpts = {MQGMO_DEFAULT};
                             /* Get Message Options      */
MQLONG   MsgBuffLen;         /* Length of message buffer */
CSQ4BCAQ MsgBuffer;          /* Message structure        */
MQLONG   DataLen;            /* Length of message        */

MQPMO    PutMsgOpts = {MQPMO_DEFAULT};
                             /* Put Message Options      */
CSQ4BQRM PutBuffer;          /* Message structure        */
MQLONG   PutBuffLen = sizeof(PutBuffer);
                             /* Length of message buffer */
.
.
.
void Process_Query(void)
   {
   /*                                                     */
   /* Build the reply message                             */
   /*                                                     */
```

```
.
.
.
   /*                                                  */
   /* Set the object descriptor, message descriptor and  */
   /* put message options to the values required to      */
   /* create the reply message.                        */
   /*                                                  */
   strncpy(ObjDesc.ObjectName, MsgDesc.ReplyToQ,
           MQ_Q_NAME_LENGTH);
   strncpy(ObjDesc.ObjectQMgrName, MsgDesc.ReplyToQMgr,
           MQ_Q_MGR_NAME_LENGTH);
   MsgDesc.MsgType = MQMT_REPLY;
   MsgDesc.Report  = MQRO_NONE;
   memset(MsgDesc.ReplyToQ, ' ', MQ_Q_NAME_LENGTH);
   memset(MsgDesc.ReplyToQMgr, ' ', MQ_Q_MGR_NAME_LENGTH);
   memcpy(MsgDesc.MsgId, MQMI_NONE, sizeof(MsgDesc.MsgId));
   PutMsgOpts.Options = MQPMO_SYNCPOINT +
                        MQPMO_PASS_IDENTITY_CONTEXT;
   PutMsgOpts.Context = Hobj_CheckQ;
   PutBuffLen = sizeof(PutBuffer);
   MQPUT1(Hconn,
          &ObjDesc,
          &MsgDesc,
          &PutMsgOpts,
          PutBuffLen,
          &PutBuffer,
          &CompCode,
          &Reason);

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

Getting a message

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

This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
#include "cmqc.h"
.
.
.
#define BUFFERLENGTH 80
.
.
.
int  main(int argc, char *argv[] )
   {
   /*                                                  */
   /*     Variables for MQ calls                       */
   /*                                                  */
   MQHCONN Hconn ;            /* Connection handle       */
   MQLONG  CompCode;          /* Completion code         */
   MQLONG  Reason;            /* Qualifying reason       */
   MQHOBJ  Hobj;              /* Object handle           */
   MQMD    MsgDesc  = { MQMD_DEFAULT };
```

```
                              /* Message descriptor      */
   MQLONG  DataLength ;       /* Length of the message   */
   MQCHAR  Buffer[BUFFERLENGTH+1];
                              /* Area for message data   */
   MQGMO   GetMsgOpts = { MQGMO_DEFAULT };
                              /* Options which control   */
                              /*  the MQGET call         */
   MQLONG  BufferLength = BUFFERLENGTH ;
                              /* Length of buffer        */
.
   /*     No need to change the message descriptor       */
   /*     (MQMD) control block because initialization    */
   /*     default sets all the fields.                   */
   /*                                                    */
   /*     Initialize the get message options (MQGMO)     */
   /*     control block (the copy file initializes all   */
   /*     the other fields).                             */
   /*                                                    */
   GetMsgOpts.Options = MQGMO_NO_WAIT      +
                        MQGMO_BROWSE_FIRST +
                        MQGMO_ACCEPT_TRUNCATED_MSG;
   /*                                                    */
   /* Get the first message.                             */
   /*   Test for the output of the call is carried out   */
   /*   in the 'for' loop.                               */
   /*                                                    */
   MQGET(Hconn,
         Hobj,
         &MsgDesc,
         &GetMsgOpts,
         BufferLength,
         Buffer,
         &DataLength,
         &CompCode,
         &Reason);
   /*                                                    */
   /* Process the message and get the next message,      */
   /* until no messages remaining.                       */
.
   /*     If the call fails for any other reason,        */
   /*     print an error message showing the completion  */
   /*     code and reason code.                          */
   /*                                                    */
   if ( (CompCode == MQCC_FAILED) &&
       (Reason   == MQRC_NO_MSG_AVAILABLE) )
      {
.
      }
   else
      {
      sprintf(pBuff, MESSAGE_4_E,
              ERROR_IN_MQGET, CompCode, Reason);
      PrintLine(pBuff);
      RetCode = CSQ4_ERROR;
      }
.
   } /* end of main */
```

Getting a message using the wait option

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

This code accepts truncated messages. This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
.
.
.
MQLONG   Hconn;              /* Connection handle        */
MQHOBJ   Hobj_CheckQ;        /* Object handle            */
MQLONG   CompCode;           /* Completion code          */
MQLONG   Reason;             /* Qualifying reason        */
MQOD     ObjDesc    = {MQOD_DEFAULT};
                             /* Object descriptor        */
MQMD     MsgDesc    = {MQMD_DEFAULT};
                             /* Message descriptor       */
MQLONG   OpenOptions;
                             /* Control the MQOPEN call  */
MQGMO    GetMsgOpts = {MQGMO_DEFAULT};
                             /* Get Message Options      */
MQLONG   MsgBuffLen;         /* Length of message buffer */
CSQ4BCAQ MsgBuffer;          /* Message structure        */
MQLONG   DataLen;            /* Length of message        */
.
.
.
void main(void)
   {
.
.
.
   /*                                                     */
   /* Initialize options and open the queue for input     */
   /*                                                     */
.
.
.
      /*                                                  */
      /* Get and process messages                         */
      /*                                                  */
      GetMsgOpts.Options = MQGMO_WAIT +
                           MQGMO_ACCEPT_TRUNCATED_MSG +
                           MQGMO_SYNCPOINT;
      GetMsgOpts.WaitInterval = WAIT_INTERVAL;
      MsgBuffLen = sizeof(MsgBuffer);
      memcpy(MsgDesc.MsgId, MQMI_NONE,
             sizeof(MsgDesc.MsgId));
      memcpy(MsgDesc.CorrelId, MQCI_NONE,
             sizeof(MsgDesc.CorrelId));
      /*                                                  */
      /* Make the first MQGET call outside the loop       */
      /*                                                  */
      MQGET(Hconn,
            Hobj_CheckQ,
            &MsgDesc,
            &GetMsgOpts,
            MsgBuffLen,
            &MsgBuffer,
            &DataLen,
            &CompCode,
            &Reason);
.
.
.
      /*                                                  */
      /* Test the output of the MQGET call.  If the call  */
      /* failed, send an error message showing the        */
      /* completion code and reason code, unless the      */
      /* reason code is NO_MSG AVAILABLE.                 */
      /*                                                  */
      if (Reason != MQRC_NO_MSG_AVAILABLE)
         {
         strncpy(TS_Operation, "MQGET", sizeof(TS_Operation));
         strncpy(TS_ObjName, ObjDesc.ObjectName,
                 MQ_Q_NAME_LENGTH);
```

```
        Record_Call_Error();
        }
⋮
```

Getting a message using signaling

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

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

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

     /*--------------------------*/
     /*  Set up GMO structure.    */
     /*--------------------------*/
     memset(&GetMsgOpts,'\0',sizeof(GetMsgOpts));
     memcpy(GetMsgOpts.StrucId, MQGMO_STRUC_ID,
            sizeof(GetMsgOpts.StrucId);
     GetMsgOpts.Version      = MQGMO_VERSION_1;
     GetMsgOpts.WaitInterval = 1000;
     GetMsgOpts.Options      = MQGMO_SET_SIGNAL +
                               MQGMO_BROWSE_FIRST;
     q_ecb              = 0;
     GetMsgOpts.Signal1      = &q_ecb;
     /*--------------------------*/
     /*  Set up MD structure.     */
     /*--------------------------*/
     memset(&MsgDesc,'\0',sizeof(MsgDesc));
     memcpy(MsgDesc.StrucId, MQMD_STRUC_ID,
            sizeof(MsgDesc.StrucId);
     MsgDesc.Version  = MQMD_VERSION_1;
     MsgDesc.Report   = MQRO_NONE;
     memcpy(MsgDesc.MsgId,MQMI_NONE,
            sizeof(MsgDesc.MsgId));
     memcpy(MsgDesc.CorrelId,MQCI_NONE,
            sizeof(MsgDesc.CorrelId));
    /*-------------------------------------------------*/
    /*  Issue the MQGET call.                          */
    /*-------------------------------------------------*/
     BufferLength = sizeof(message_buffer);
     signal_sw   = 0;

     MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
           BufferLength, message_buffer, &DataLength,
           &CompCode, &Reason);
    /*-------------------------------------*/
    /*  Check completion and reason codes. */
    /*-------------------------------------*/
```

```
 switch (CompCode)
     {
      case (MQCC_OK):          /* Message retrieved  */
          break;
      case (MQCC_WARNING):
          switch (Reason)
          {
           case (MQRC_SIGNAL_REQUEST_ACCEPTED):
               signal_sw = 1;
               break;
           default:
               break;   /* Perform error processing  */
          }
          break;
      case (MQCC_FAILED):
          switch (Reason)
          {
           case (MQRC_Q_MGR_NOT_AVAILABLE):
           case (MQRC_CONNECTION_BROKEN):
           case (MQRC_Q_MGR_STOPPING):
               break;
           default:
               break;  /* Perform error processing.  */
          }
          break;
      default:
          break;       /* Perform error processing.  */
     }
/*-----------------------------------------------------*/
/* If the SET_SIGNAL was accepted, set up a loop to  */
/* check whether a message has arrived at one second */
/* intervals. The loop ends if a message arrives or  */
/* the wait interval specified in the MQGMO          */
/* structure has expired.                            */
/*                                                   */
/* If a message arrives on the queue, another MQGET  */
/* must be issued to retrieve the message. If other  */
/* MQM calls have been made in the intervening       */
/* period, this may necessitate reinitializing the   */
/* MQMD and MQGMO structures.                        */
/* In this code, no intervening calls                */
/* have been made, so the only change required to    */
/* the structures is to specify MQGMO_NO_WAIT,       */
/* since we now know the message is there.           */
/*                                                   */
/* This code uses the EXEC CICS DELAY command to     */
/* suspend the program for a second. A batch program */
/* may achieve the same effect by calling an         */
/* assembler language subroutine which issues a      */
/* z/OS STIMER macro.                                */
/*-----------------------------------------------------*/
if (signal_sw == 1)
   {
    endloop = 0;
    do
      {
       EXEC CICS DELAY FOR HOURS(0) MINUTES(0) SECONDS(1);
       work_ecb = q_ecb & mask;
       switch (work_ecb)
           {
            case (MQEC_MSG_ARRIVED):
                endloop = 1;
                mqgmo_options = MQGMO_NO_WAIT;
                MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
                      BufferLength, message_buffer,
                      &DataLength, &CompCode, &Reason);
                if (CompCode != MQCC_OK)
```

```
                        ;          /* Perform error processing. */
                    break;
                case (MQEC_WAIT_INTERVAL_EXPIRED):
                case (MQEC_WAIT_CANCELED):
                    endloop = 1;
                    break;
                default:
                    break;
                }
            } while (endloop == 0);
        }
    return;
}
```

Inquiring about the attributes of an object

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

This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

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

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

   {
   /*      Declare local variables                        */
   /*                                                     */
   MQLONG  SelectorCount = NUMBEROFSELECTORS;
                                /* Number of selectors  */
   MQLONG  IntAttrCount  = NUMBEROFSELECTORS;
                                /* Number of int attrs  */
   MQLONG  CharAttrLength = 0;
                     /* Length of char attribute buffer  */
   MQCHAR *CharAttrs ;
                      /* Character attribute buffer      */
   MQLONG  SelectorsTable[NUMBEROFSELECTORS];
                                /* attribute selectors  */
   MQLONG  IntAttrsTable[NUMBEROFSELECTORS];
                                /* integer attributes   */
   MQLONG  CompCode;            /* Completion code      */
   MQLONG  Reason;              /* Qualifying reason    */
   /*                                                     */
   /*     Open the queue.  If successful, do the inquire */
   /*     call.                                          */
   /*                                                     */
      /*                                                  */
      /*   Initialize the variables for the inquire      */
      /*   call:                                         */
      /*    - Set SelectorsTable to the attributes whose */
      /*      status is                                   */
      /*       required                                   */
      /*    - All other variables are already set        */
      /*                                                  */
      SelectorsTable[0] = MQIA_INHIBIT_GET;
      SelectorsTable[1] = MQIA_INHIBIT_PUT;
      /*                                                  */
```

```
/*   Issue the inquire call                          */
/*     Test the output of the inquire call. If the */
/*     call failed, display an error message       */
/*     showing the completion code and reason code,*/
/*     otherwise display the status of the         */
/*     INHIBIT-GET and INHIBIT-PUT attributes      */
/*                                                 */
MQINQ(Hconn,
      *pHobj,
      SelectorCount,
      SelectorsTable,
      IntAttrCount,
      IntAttrsTable,
      CharAttrLength,
      CharAttrs,
      &CompCode,
      &Reason);
if (CompCode != MQCC_OK)
   {
   sprintf(Message, MESSAGE_4_E,
           ERROR_IN_MQINQ, CompCode, Reason);
   SetMsg(Message);
   }
else
   {
     /* Process the changes */
   } /* end if CompCode */
```

Setting the attributes of a queue

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

This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

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

const MQHCONN Hconn = MQHC_DEF_HCONN;

static void InhibitGetAndPut(char   *Message,
                             PMQHOBJ pHobj,
                             char   *Object)
   {
   /*                                                 */
   /*      Declare local variables                    */
   /*                                                 */
   MQLONG  SelectorCount = NUMBEROFSELECTORS;
                              /* Number of selectors  */
   MQLONG  IntAttrCount  = NUMBEROFSELECTORS;
                              /* Number of int attrs  */
   MQLONG  CharAttrLength = 0;
                    /* Length of char attribute buffer  */
   MQCHAR *CharAttrs ;
                    /* Character attribute buffer       */
   MQLONG  SelectorsTable[NUMBEROFSELECTORS];
                              /* attribute selectors  */
   MQLONG  IntAttrsTable[NUMBEROFSELECTORS];
                              /* integer attributes   */
   MQLONG  CompCode;          /* Completion code      */
   MQLONG  Reason;            /* Qualifying reason    */
```

```
.
.
.
   /*                                                          */
   /*     Open the queue.  If successful, do the               */
   /*     inquire call.                                        */
   /*                                                          */
.
.
.
      /*                                                       */
      /*   Initialize the variables for the set call:          */
      /*    - Set SelectorsTable to the attributes to be       */
      /*      set                                              */
      /*    - Set IntAttrsTable to the required status         */
      /*    - All other variables are already set              */
      /*                                                       */
      SelectorsTable[0] = MQIA_INHIBIT_GET;
      SelectorsTable[1] = MQIA_INHIBIT_PUT;
      IntAttrsTable[0]  = MQQA_GET_INHIBITED;
      IntAttrsTable[1]  = MQQA_PUT_INHIBITED;
.
.
.
      /*                                                       */
      /*   Issue the set call.                                 */
      /*     Test the output of the set call.  If the          */
      /*     call fails, display an error message              */
      /*     showing the completion code and reason            */
      /*     code; otherwise move INHIBITED to the             */
      /*     relevant screen map fields                        */
      /*                                                       */
      MQSET(Hconn,
            *pHobj,
            SelectorCount,
            SelectorsTable,
            IntAttrCount,
            IntAttrsTable,
            CharAttrLength,
            CharAttrs,
            &CompCode,
            &Reason);
      if (CompCode != MQCC_OK)
         {
         sprintf(Message, MESSAGE_4_E,
                 ERROR_IN_MQSET, CompCode, Reason);
         SetMsg(Message);
         }
      else
         {
           /* Process the changes */
         } /* end if CompCode */
```

Retrieving status information with MQSTAT

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

This extract is taken from the Calling MQSTAT sample application (program amqsapt0) supplied with WebSphere MQ for Windows systems. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
/*****************************************************************/
/*                                                               */
/* Program name: AMQSAPT0                                        */
/*                                                               */
/* Description: Sample C program that asynchronously puts messages  */
```

```
/*              to a message queue (example using MQPUT & MQSTAT).  */
/*                                                                  */
/* Licensed Materials - Property of IBM                             */
/*                                                                  */
/* 63H9336                                                          */
/* (c) Copyright IBM Corp. 2006 All Rights Reserved.                */
/*                                                                  */
/* US Government Users Restricted Rights - Use, duplication or      */
/* disclosure restricted by GSA ADP Schedule Contract with          */
/* IBM Corp.                                                        */
/*                                                                  */
/********************************************************************/
/*                                                                  */
/* Function:                                                        */
/*                                                                  */
/*   AMQSAPT0 is a sample C program to put messages on a message    */
/*   queue with asynchronous response option, querying the success  */
/*   of the put operations with MQSTAT.                             */
/*                                                                  */
/*      -- messages are sent to the queue named by the parameter    */
/*                                                                  */
/*      -- gets lines from StdIn, and adds each to target           */
/*         queue, taking each line of text as the content           */
/*         of a datagram message; the sample stops when a null      */
/*         line (or EOF) is read.                                   */
/*         New-line characters are removed.                         */
/*         If a line is longer than 99 characters it is broken up   */
/*         into 99-character pieces. Each piece becomes the         */
/*         content of a datagram message.                           */
/*         If the length of a line is a multiple of 99 plus 1       */
/*         e.g. 199, the last piece will only contain a new-line    */
/*         character so will terminate the input.                   */
/*                                                                  */
/*      -- writes a message for each MQI reason other than          */
/*         MQRC_NONE; stops if there is a MQI completion code       */
/*         of MQCC_FAILED                                           */
/*                                                                  */
/*      -- summarizes the overall success of the put operations     */
/*         through a call to MQSTAT to query MQSTAT_TYPE_ASYNC_ERROR*/
/*                                                                  */
/*    Program logic:                                                */
/*         MQOPEN target queue for OUTPUT                           */
/*         while end of input file not reached,                     */
/*         .  read next line of text                                */
/*         .  MQPUT datagram message with text line as data         */
/*         MQCLOSE target queue                                     */
/*         MQSTAT connection                                        */
/*                                                                  */
/*                                                                  */
/********************************************************************/
/*                                                                  */
/*   AMQSAPT0 has the following parameters                          */
/*       required:                                                  */
/*                 (1) The name of the target queue                 */
/*       optional:                                                  */
/*                 (2) Queue manager name                           */
/*                 (3) The open options                             */
/*                 (4) The close options                            */
/*                 (5) The name of the target queue manager         */
/*                 (6) The name of the dynamic queue                */
/*                                                                  */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
   /* includes for MQI */
#include <cmqc.h>
```

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

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

  printf("Sample AMQSAPT0 start\n");
  if (argc < 2)
  {
    printf("Required parameter missing - queue name\n");
    exit(99);
  }

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

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

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

  if (argc > 6)
  {
    strncpy(od.DynamicQName, argv[6], (size_t) MQ_Q_NAME_LENGTH);
```

```
  printf("dynamic queue name is %s\n", od.DynamicQName);
}

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

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

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

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

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

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

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

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

while (CompCode != MQCC_FAILED)
{
```

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 6. COBOL examples

The examples in this appendix are taken from the WebSphere MQ for z/OS sample applications. They are applicable to all platforms, except where noted.

The examples in this appendix demonstrate the following techniques:

Connecting to a queue manager

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

This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

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

```
*
     CALL 'MQCONN' USING W02-MQM
                         W03-HCONN
                         W03-COMPCODE
                         W03-REASON.
*
*    Test the output of the connect call.  If the call
*    fails, print an error message showing the
*    completion code and reason code.
*
     IF (W03-COMPCODE NOT = MQCC-OK) THEN
:
     END-IF.
:
```

Disconnecting from a queue manager

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

The variables used in this code extract are those that were set in "Connecting to a queue manager" on page 525. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

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

Creating a dynamic queue

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

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
:
* -------------------------------------------------------*
 WORKING-STORAGE SECTION.
* -------------------------------------------------------*
*
*    W02 - Queues processed in this program
*
 01  W02-MODEL-QNAME        PIC X(48) VALUE
```

```
     'CSQ4SAMP.B1.MODEL                          '.
 01  W02-NAME-PREFIX        PIC X(48) VALUE
     'CSQ4SAMP.B1.*                              '.
 01  W02-TEMPORARY-Q         PIC X(48).
*
*    W03 - MQM API fields
*
 01  W03-HCONN       PIC S9(9) BINARY VALUE ZERO.
 01  W03-OPTIONS     PIC S9(9) BINARY.
 01  W03-HOBJ        PIC S9(9) BINARY.
 01  W03-COMPCODE    PIC S9(9) BINARY.
 01  W03-REASON      PIC S9(9) BINARY.
*
*    API control blocks
*
 01  MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
*
*    CMQV contains constants (for setting or testing
*    field values) and return codes (for testing the
*    result of a call)
*
 01  MQM-CONSTANTS.
 COPY CMQV SUPPRESS.
* ---------------------------------------------------------*
 PROCEDURE DIVISION.
* ---------------------------------------------------------*
:
* ---------------------------------------------------------*
 OPEN-TEMP-RESPONSE-QUEUE SECTION.
* ---------------------------------------------------------*
*
*  This section creates a temporary dynamic queue
*  using a model queue
*
* ---------------------------------------------------------*
*
* Change three fields in the Object Descriptor (MQOD)
* control block.  (MQODV initializes the other fields)
*
     MOVE MQOT-Q           TO MQOD-OBJECTTYPE.
     MOVE W02-MODEL-QNAME  TO MQOD-OBJECTNAME.
     MOVE W02-NAME-PREFIX  TO MQOD-DYNAMICQNAME.
*
     COMPUTE W03-OPTIONS = MQOO-INPUT-EXCLUSIVE.
*
     CALL 'MQOPEN' USING W03-HCONN
                         MQOD
                         W03-OPTIONS
                         W03-HOBJ-MODEL
                         W03-COMPCODE
                         W03-REASON.
*
     IF W03-COMPCODE NOT = MQCC-OK
          MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
          MOVE W03-COMPCODE  TO M01-MSG4-COMPCODE
          MOVE W03-REASON    TO M01-MSG4-REASON
          MOVE M01-MESSAGE-4 TO M00-MESSAGE
     ELSE
          MOVE MQOD-OBJECTNAME TO W02-TEMPORARY-Q
     END-IF.
*
 OPEN-TEMP-RESPONSE-QUEUE-EXIT.
*
*    Return to performing section.
```

```
*
     EXIT.
     EJECT
*
```

Opening an existing queue

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

This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
:
* ------------------------------------------------------*
 WORKING-STORAGE SECTION.
* ------------------------------------------------------*
*
*    W01 - Fields derived from the command area input
*
 01  W01-OBJECT                  PIC X(48).
*
*    W02 - MQM API fields
*
 01  W02-HCONN        PIC S9(9) BINARY VALUE ZERO.
 01  W02-OPTIONS      PIC S9(9) BINARY.
 01  W02-HOBJ         PIC S9(9) BINARY.
 01  W02-COMPCODE     PIC S9(9) BINARY.
 01  W02-REASON       PIC S9(9) BINARY.
*
*    CMQODV defines the object descriptor (MQOD)
*
 01  MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
*
* CMQV contains constants (for setting or testing
* field values) and return codes (for testing the
* result of a call)
*
 01  MQM-CONSTANTS.
 COPY CMQV SUPPRESS.
* ------------------------------------------------------*
 E-OPEN-QUEUE SECTION.
* ------------------------------------------------------*
*                                                       *
* This section opens the queue                          *
*
*    Initialize the Object Descriptor (MQOD) control
*    block
*    (The copy file initializes the remaining fields.)
*
     MOVE MQOT-Q         TO MQOD-OBJECTTYPE.
     MOVE W01-OBJECT     TO MQOD-OBJECTNAME.
*
*    Initialize W02-OPTIONS to open the queue for both
*    inquiring about and setting attributes
*
     COMPUTE W02-OPTIONS = MQOO-INQUIRE + MQOO-SET.

*
*    Open the queue
*
     CALL 'MQOPEN' USING W02-HCONN
```

```
                        MQOD
                        W02-OPTIONS
                        W02-HOBJ
                        W02-COMPCODE
                        W02-REASON.
*
*    Test the output from the open
*
*    If the completion code is not OK, display a
*    separate error message for each of the following
*    errors:
*
*  Q-MGR-NOT-AVAILABLE - MQM is not available
*  CONNECTION-BROKEN   - MQM is no longer connected to CICS
*  UNKNOWN-OBJECT-NAME - The queue does not exist
*  NOT-AUTHORIZED      - The user is not authorized to open
*                        the queue
*
* For any other error, display an error message
* showing the completion and reason codes
*
  IF W02-COMPCODE NOT = MQCC-OK
     EVALUATE TRUE
*
       WHEN W02-REASON = MQRC-Q-MGR-NOT-AVAILABLE
            MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
       WHEN W02-REASON = MQRC-CONNECTION-BROKEN
            MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
       WHEN W02-REASON = MQRC-UNKNOWN-OBJECT-NAME
            MOVE M01-MESSAGE-2 TO M00-MESSAGE
*
       WHEN W02-REASON = MQRC-NOT-AUTHORIZED
            MOVE M01-MESSAGE-3 TO M00-MESSAGE
*
       WHEN OTHER
            MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
            MOVE W02-COMPCODE  TO M01-MSG4-COMPCODE
            MOVE W02-REASON    TO M01-MSG4-REASON
            MOVE M01-MESSAGE-4 TO M00-MESSAGE
        END-EVALUATE
     END-IF.
 E-EXIT.
*
*    Return to performing section
*
     EXIT.
     EJECT
```

Closing a queue

This example demonstrates how to use the MQCLOSE call.

The variables used in this code extract are those that were set in "Connecting to a queue manager" on page 525. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395.

```
⋮
*
*    Close the queue
*
```

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

Putting a message using MQPUT

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

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

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

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

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call.

This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
:
* -----------------------------------------------------*
 WORKING-STORAGE SECTION.
* -----------------------------------------------------*
*
*    W03 - MQM API fields
*
 01  W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
 01  W03-OPTIONS        PIC S9(9) BINARY.
 01  W03-COMPCODE       PIC S9(9) BINARY.
 01  W03-REASON         PIC S9(9) BINARY.
 01  W03-BUFFLEN        PIC S9(9) BINARY.
*
 01  W03-PUT-BUFFER.
     05 W03-CSQ4BQRM.
     COPY CSQ4VB4.
```

```
*
*    API control blocks
*
 01  MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
 01  MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 01  MQM-PUT-MESSAGE-OPTIONS.
     COPY CMQPMOV.
*
* CMQV contains constants (for filling in the
* control blocks) and return codes (for testing
* the result of a call).
*
 01  MQM-MQV.
 COPY CMQV SUPPRESS.
* ----------------------------------------------------------*
 PROCEDURE DIVISION.
* ----------------------------------------------------------*
:
*    Get the request message.
:
* ----------------------------------------------------------*
 PROCESS-QUERY SECTION.
* ----------------------------------------------------------*
:
*    Build the reply message.
:
*
* Set the object descriptor, message descriptor and
* put-message options to the values required to create
* the message.
* Set the length of the message.
*
  MOVE MQMD-REPLYTOQ    TO MQOD-OBJECTNAME.
  MOVE MQMD-REPLYTOQMGR TO MQOD-OBJECTQMGRNAME.
  MOVE MQMT-REPLY       TO MQMD-MSGTYPE.
  MOVE SPACES           TO MQMD-REPLYTOQ.
  MOVE SPACES           TO MQMD-REPLYTOQMGR.
  MOVE LOW-VALUES       TO MQMD-MSGID.
  COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
                          MQPMO-PASS-IDENTITY-CONTEXT.
  MOVE W03-HOBJ-CHECKQ  TO MQPMO-CONTEXT.
  MOVE LENGTH OF CSQ4BQRM-MSG TO W03-BUFFLEN.
*
     CALL 'MQPUT1' USING W03-HCONN
                         MQOD
                         MQMD
                         MQPMO
                         W03-BUFFLEN
                         W03-PUT-BUFFER
                         W03-COMPCODE
                         W03-REASON.
     IF W03-COMPCODE NOT = MQCC-OK
         MOVE 'MQPUT1'          TO M02-OPERATION
         MOVE MQOD-OBJECTNAME   TO M02-OBJECTNAME
         PERFORM RECORD-CALL-ERROR
         PERFORM FORWARD-MSG-TO-DLQ
     END-IF.
*
```

Getting a message

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

This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
:
* -------------------------------------------------------*
 WORKING-STORAGE SECTION.
* -------------------------------------------------------*
*
*    W03 - MQM API fields
*
 01  W03-HCONN            PIC S9(9) BINARY VALUE ZERO.
 01  W03-HOBJ-RESPONSE    PIC S9(9) BINARY.
 01  W03-OPTIONS          PIC S9(9) BINARY.
 01  W03-BUFFLEN          PIC S9(9) BINARY.
 01  W03-DATALEN          PIC S9(9) BINARY.
 01  W03-COMPCODE         PIC S9(9) BINARY.
 01  W03-REASON           PIC S9(9) BINARY.
*
 01  W03-GET-BUFFER.
     05 W03-CSQ4BAM.
     COPY CSQ4VB2.
*
*    API control blocks
*
 01  MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 01  MQM-GET-MESSAGE-OPTIONS.
     COPY CMQGMOV.
*
*    MQV contains constants (for filling in the
*    control blocks) and return codes (for testing
*    the result of a call).
*
 01  MQM-CONSTANTS.
     COPY CMQV SUPPRESS.
* -------------------------------------------------------*
 A-MAIN SECTION.
* -------------------------------------------------------*
:
*    Open response queue.
:
* -------------------------------------------------------*
 PROCESS-RESPONSE-SCREEN SECTION.
* -------------------------------------------------------*
*                                                        *
*  This section gets a message from the response queue.  *
*                                                        *
*  When a correct response is received, it is            *
*  transferred to the map for display; otherwise         *
*  an error message is built.                            *
*                                                        *
* -------------------------------------------------------*

*
*    Set get-message options
*
  COMPUTE MQGMO-OPTIONS = MQGMO-SYNCPOINT +
                          MQGMO-ACCEPT-TRUNCATED-MSG +
                          MQGMO-NO-WAIT.
*
* Set msgid and correlid in MQMD to nulls so that any
* message will qualify.
* Set length to available buffer length.
*
```

```
     MOVE MQMI-NONE TO MQMD-MSGID.
     MOVE MQCI-NONE TO MQMD-CORRELID.
     MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
     CALL 'MQGET' USING W03-HCONN
                        W03-HOBJ-RESPONSE
                        MQMD
                        MQGMO
                        W03-BUFFLEN
                        W03-GET-BUFFER
                        W03-DATALEN
                        W03-COMPCODE
                        W03-REASON.
     EVALUATE TRUE
         WHEN W03-COMPCODE NOT = MQCC-FAILED
:
*            Process the message
:
         WHEN (W03-COMPCODE = MQCC-FAILED AND
               W03-REASON = MQRC-NO-MSG-AVAILABLE)
                 MOVE M01-MESSAGE-9 TO M00-MESSAGE
                 PERFORM CLEAR-RESPONSE-SCREEN
*
         WHEN OTHER
             MOVE 'MQGET  '     TO M01-MSG4-OPERATION
             MOVE W03-COMPCODE  TO M01-MSG4-COMPCODE
             MOVE W03-REASON    TO M01-MSG4-REASON
             MOVE M01-MESSAGE-4 TO M00-MESSAGE
             PERFORM CLEAR-RESPONSE-SCREEN
     END-EVALUATE.
```

Getting a message using the wait option

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

This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
:
* -------------------------------------------------------*
 WORKING-STORAGE SECTION.
* -------------------------------------------------------*
*
*    W00 - General work fields
*
 01  W00-WAIT-INTERVAL   PIC S9(09) BINARY VALUE 30000.
*
*    W03 - MQM API fields
*
 01  W03-HCONN           PIC S9(9) BINARY VALUE ZERO.
 01  W03-OPTIONS         PIC S9(9) BINARY.
 01  W03-HOBJ-CHECKQ     PIC S9(9) BINARY.
 01  W03-COMPCODE        PIC S9(9) BINARY.
 01  W03-REASON          PIC S9(9) BINARY.
 01  W03-DATALEN         PIC S9(9) BINARY.
 01  W03-BUFFLEN         PIC S9(9) BINARY.
*
 01  W03-MSG-BUFFER.
     05 W03-CSQ4BCAQ.
     COPY CSQ4VB3.
*
```

```
*    API control blocks
*
 01  MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 01  MQM-GET-MESSAGE-OPTIONS.
     COPY CMQGMOV.
*
*    CMQV contains constants (for filling in the
*    control blocks) and return codes (for testing
*    the result of a call).
*
 01  MQM-MQV.
 COPY CMQV SUPPRESS.
* -------------------------------------------------------*
 PROCEDURE DIVISION.
* -------------------------------------------------------*
:
*    Open input queue.
:
*
*    Get and process messages.
*
  COMPUTE MQGMO-OPTIONS = MQGMO-WAIT +
                          MQGMO-ACCEPT-TRUNCATED-MSG +
                          MQGMO-SYNCPOINT.
  MOVE LENGTH OF W03-MSG-BUFFER TO W03-BUFFLEN.
  MOVE W00-WAIT-INTERVAL TO MQGMO-WAITINTERVAL.
  MOVE MQMI-NONE TO MQMD-MSGID.
  MOVE MQCI-NONE TO MQMD-CORRELID.
*
*    Make the first MQGET call outside the loop.
*
     CALL 'MQGET' USING W03-HCONN
                        W03-HOBJ-CHECKQ
                        MQMD
                        MQGMO
                        W03-BUFFLEN
                        W03-MSG-BUFFER
                        W03-DATALEN
                        W03-COMPCODE
                        W03-REASON.
*
*    Test the output of the MQGET call using the
*    PERFORM loop that follows.
*
*    Perform whilst no failure occurs
*      - process this message
*      - reset the call parameters
*      - get another message
*    End-perform
*
:
*
*    Test the output of the MQGET call.  If the call
*    fails, send an error message showing the
*    completion code and reason code, unless the
*    completion code is NO-MSG-AVAILABLE.
*
     IF (W03-COMPCODE NOT = MQCC-FAILED) OR
        (W03-REASON NOT = MQRC-NO-MSG-AVAILABLE)
         MOVE 'MQGET '          TO M02-OPERATION
         MOVE MQOD-OBJECTNAME   TO M02-OBJECTNAME
               PERFORM RECORD-CALL-ERROR
```

```
     END-IF.
:
```

Getting a message using signaling

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

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

```
:
* -------------------------------------------------------*
 WORKING-STORAGE SECTION.
* -------------------------------------------------------*
*
*    W00 - General work fields
:
 01  W00-WAIT-INTERVAL    PIC S9(09) BINARY VALUE 30000.
*
*    W03 - MQM API fields
*
 01  W03-HCONN            PIC S9(9) BINARY VALUE ZERO.
 01  W03-HOBJ-REPLYQ      PIC S9(9) BINARY.
 01  W03-COMPCODE         PIC S9(9) BINARY.
 01  W03-REASON           PIC S9(9) BINARY.
 01  W03-DATALEN          PIC S9(9) BINARY.
 01  W03-BUFFLEN          PIC S9(9) BINARY.
:
 01  W03-GET-BUFFER.
     05 W03-CSQ4BQRM.
     COPY CSQ4VB4.
*
     05 W03-CSQ4BIIM REDEFINES W03-CSQ4BQRM.
     COPY CSQ4VB1.
*
     05 W03-CSQ4BPGM REDEFINES W03-CSQ4BIIM.
     COPY CSQ4VB5.
:
*    API control blocks
*
 01  MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 01  MQM-GET-MESSAGE-OPTIONS.
     COPY CMQGMOV.
:
*    MQV contains constants (for filling in the
*    control blocks) and return codes (for testing
*    the result of a call).
*
 01  MQM-MQV.
 COPY CMQV SUPPRESS.
* -------------------------------------------------------*
 LINKAGE SECTION.
* -------------------------------------------------------*
 01  L01-ECB-ADDR-LIST.
     05  L01-ECB-ADDR1        POINTER.
     05  L01-ECB-ADDR2        POINTER.
*
 01  L02-ECBS.
     05  L02-INQUIRY-ECB1     PIC S9(09) BINARY.
     05  L02-REPLY-ECB2       PIC S9(09) BINARY.
 01  REDEFINES L02-ECBS.
```

```
     05                      PIC  X(02).
     05  L02-INQUIRY-ECB1-CC  PIC S9(04) BINARY.
     05                      PIC  X(02).
     05  L02-REPLY-ECB2-CC    PIC S9(04) BINARY.
*
* ------------------------------------------------------*
 PROCEDURE DIVISION.
* ------------------------------------------------------*
:
* Initialize variables, open queues, set signal on
* inquiry queue.
:
* ------------------------------------------------------*
 PROCESS-SIGNAL-ACCEPTED SECTION.
* ------------------------------------------------------*
*  This section gets a message with signal.  If a       *
*  message is received, process it.  If the signal      *
*  is set or is already set, the program goes into      *
*  an operating system wait.                            *
*  Otherwise an error is reported and call error set.   *
* ------------------------------------------------------*
*
  PERFORM REPLYQ-GETSIGNAL.
*
  EVALUATE TRUE
      WHEN (W03-COMPCODE = MQCC-OK AND
              W03-REASON = MQRC-NONE)
          PERFORM PROCESS-REPLYQ-MESSAGE
*
      WHEN (W03-COMPCODE = MQCC-WARNING AND
              W03-REASON = MQRC-SIGNAL-REQUEST-ACCEPTED)
           OR
           (W03-COMPCODE = MQCC-FAILED AND
              W03-REASON = MQRC-SIGNAL-OUTSTANDING)
          PERFORM EXTERNAL-WAIT
*
      WHEN OTHER
          MOVE 'MQGET SIGNAL'  TO M02-OPERATION
          MOVE MQOD-OBJECTNAME TO M02-OBJECTNAME
          PERFORM RECORD-CALL-ERROR
          MOVE W06-CALL-ERROR  TO W06-CALL-STATUS
  END-EVALUATE.
*
 PROCESS-SIGNAL-ACCEPTED-EXIT.
*    Return to performing section
     EXIT.
     EJECT
*
* ------------------------------------------------------*
 EXTERNAL-WAIT SECTION.
* ------------------------------------------------------*
*  This section performs an external CICS wait on two   *
*  ECBs until at least one is posted.  It then calls    *
*  the sections to handle the posted ECB.               *
* ------------------------------------------------------*
     EXEC CICS WAIT EXTERNAL
         ECBLIST(W04-ECB-ADDR-LIST-PTR)
         NUMEVENTS(2)
     END-EXEC.
*
* At least one ECB must have been posted to get to this
* point.  Test which ECB has been posted and perform
* the appropriate section.
*
     IF L02-INQUIRY-ECB1 NOT = 0
         PERFORM TEST-INQUIRYQ-ECB
```

```
     ELSE
         PERFORM TEST-REPLYQ-ECB
     END-IF.
*
 EXTERNAL-WAIT-EXIT.
*
*    Return to performing section.
*
     EXIT.
     EJECT
:
* ------------------------------------------------------*
 REPLYQ-GETSIGNAL SECTION.
* ------------------------------------------------------*
*                                                       *
* This section performs an MQGET call (in syncpoint with *
* signal) on the reply queue.  The signal field in the   *
* MQGMO is set to the address of the ECB.               *
* Response handling is done by the performing section.   *
*                                                       *
* ------------------------------------------------------*
*
     COMPUTE MQGMO-OPTIONS         =  MQGMO-SYNCPOINT +
                                      MQGMO-SET-SIGNAL.
     MOVE W00-WAIT-INTERVAL        TO MQGMO-WAITINTERVAL.
     MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
     MOVE ZEROS                    TO L02-REPLY-ECB2.
     SET MQGMO-SIGNAL1 TO ADDRESS OF L02-REPLY-ECB2.
*
*    Set msgid and correlid to nulls so that any message
*    will qualify.
*
     MOVE MQMI-NONE TO MQMD-MSGID.
     MOVE MQCI-NONE TO MQMD-CORRELID.
*
     CALL 'MQGET' USING W03-HCONN
                        W03-HOBJ-REPLYQ
                        MQMD
                        MQGMO
                        W03-BUFFLEN
                        W03-GET-BUFFER
                        W03-DATALEN
                        W03-COMPCODE
                        W03-REASON.
*
 REPLYQ-GETSIGNAL-EXIT.
*
*    Return to performing section.
*
     EXIT.
     EJECT
*
:
```

Inquiring about the attributes of an object

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

This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see “Sample programs (all platforms except z/OS)” on page 395.

```
:
* -------------------------------------------------------*
 WORKING-STORAGE SECTION.
* -------------------------------------------------------*
*
*    W02 - MQM API fields
*
 01  W02-SELECTORCOUNT    PIC S9(9) BINARY VALUE 2.
 01  W02-INTATTRCOUNT     PIC S9(9) BINARY VALUE 2.
 01  W02-CHARATTRLENGTH   PIC S9(9) BINARY VALUE ZERO.
 01  W02-CHARATTRS        PIC X     VALUE LOW-VALUES.
 01  W02-HCONN            PIC S9(9) BINARY VALUE ZERO.
 01  W02-HOBJ             PIC S9(9) BINARY.
 01  W02-COMPCODE         PIC S9(9) BINARY.
 01  W02-REASON           PIC S9(9) BINARY.
 01  W02-SELECTORS-TABLE.
     05  W02-SELECTORS    PIC S9(9) BINARY OCCURS 2 TIMES
 01  W02-INTATTRS-TABLE.
     05  W02-INTATTRS     PIC S9(9) BINARY OCCURS 2 TIMES
*
*    CMQODV defines the object descriptor (MQOD).
*
 01  MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
*
* CMQV contains constants (for setting or testing field
* values) and return codes (for testing the result of a
* call).
*
 01  MQM-CONSTANTS.
 COPY CMQV SUPPRESS.
* -------------------------------------------------------*
 PROCEDURE DIVISION.
* -------------------------------------------------------*
*
*    Get the queue name and open the queue.
*
:
*
*    Initialize the variables for the inquiry call:
*     - Set W02-SELECTORS-TABLE to the attributes whose
*    status is required
*     - All other variables are already set
*
     MOVE MQIA-INHIBIT-GET TO W02-SELECTORS(1).
     MOVE MQIA-INHIBIT-PUT TO W02-SELECTORS(2).

*
*    Inquire about the attributes.
*
     CALL 'MQINQ' USING W02-HCONN,
                        W02-HOBJ,
                        W02-SELECTORCOUNT,
                        W02-SELECTORS-TABLE,
                        W02-INTATTRCOUNT,
                        W02-INTATTRS-TABLE,
                        W02-CHARATTRLENGTH,
                        W02-CHARATTRS,
                        W02-COMPCODE,
                        W02-REASON.
*
* Test the output from the inquiry:
*
* - If the completion code is not OK, display an error
*   message showing the completion and reason codes
*
```

```
* - Otherwise, move the correct attribute status into
*   the relevant screen map fields
*
     IF W02-COMPCODE NOT = MQCC-OK
        MOVE 'MQINQ'       TO M01-MSG4-OPERATION
        MOVE W02-COMPCODE  TO M01-MSG4-COMPCODE
        MOVE W02-REASON    TO M01-MSG4-REASON
        MOVE M01-MESSAGE-4 TO M00-MESSAGE
*
     ELSE
*       Process the changes.
:
           END-IF.
:
```

Setting the attributes of a queue

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

This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with WebSphere MQ for z/OS. For the names and locations of the sample applications on other platforms, see "Sample programs (all platforms except z/OS)" on page 395

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

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

Chapter 7. System/390 assembler-language examples

The extracts in this appendix are mostly taken from the WebSphere MQ for z/OS sample applications.

The examples in this appendix demonstrate the following techniques:

Connecting to a queue manager

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

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

```
⋮
WORKAREA DSECT
*
PARMLIST CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
COMPCODE DS    F              Completion code
REASON   DS    F              Reason code
HCONN    DS    F              Connection handle
         ORG
PARMADDR DS    F              Address of parm field
PARMLEN  DS    H              Length of parm field
*
MQMNAME  DS    CL48           Queue manager name
*
*
************************************************************
*  SECTION NAME : MAINPARM                                 *
************************************************************
MAINPARM DS    0H
         MVI   MQMNAME,X'40'
         MVC   MQMNAME+1(L'MQMNAME-1),MQMNAME
*
* Space out first byte and initialize
*
*
* Code to address and verify parameters passed omitted
*
*
```

```
PARM1MVE DS    0H
         SR    R1,R3           Length of data
         LA    R4,MQMNAME      Address for target
         BCTR  R1,R0           Reduce for execute
         EX    R1,MOVEPARM     Move the data
*
*********************************************************
* EXECUTES                                              *
*********************************************************
MOVEPARM MVC   0(*-*,R4),0(R3)
*
         EJECT
*********************************************************
*  SECTION NAME : MAINCONN                              *
*********************************************************
*
*
MAINCONN DS    0H
         XC    HCONN,HCONN       Null connection handle
*
         CALL  MQCONN,                           X
               (MQMNAME,                         X
               HCONN,                            X
               COMPCODE,                         X
               REASON),                          X
               MF=(E,PARMLIST),VL
*
         LA    R0,MQCC_OK        Expected compcode
         C     R0,COMPCODE       As expected?
         BER   R6                Yes .. return to caller
*
         MVC   INF4_TYP,=CL10'CONNECT   '
         BAL   R7,ERRCODE             Translate error
         LA    R0,8                   Set exit code
         ST    R0,EXITCODE            to 8
         B     ENDPROG                End the program
*
```

Disconnecting from a queue manager

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

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

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

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

Creating a dynamic queue

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

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

```
:
*
*    R5 = WORK REGISTER.
*
OPEN     DS   0H
*
         MVC  WOD_AREA,MQOD_AREA INITIALIZE WORKING VERSION OF
*                                 MQOD WITH DEFAULTS
         MVC  WOD_OBJECTNAME,MOD_Q    COPY IN THE MODEL Q NAME
         MVC  WOD_DYNAMICQNAME,DYN_Q  COPY IN THE DYNAMIC Q NAME
         L    R5,=AL4(MQOO_OUTPUT)    OPEN FOR OUTPUT AND
         A    R5,=AL4(MQOO_INQUIRE)   INQUIRE
         ST   R5,OPTIONS
*
* ISSUE MQI OPEN REQUEST USING REENTRANT
* FORM OF CALL MACRO
*
         CALL MQOPEN,                           X
               (HCONN,                          X
               WOD,                             X
               OPTIONS,                         X
               HOBJ,                            X
               COMPCODE,                        X
               REASON),VL,MF=(E,CALLLST)
*
         LA  R5,MQCC_OK               CHECK THE COMPLETION CODE
         C   R5,COMPCODE              FROM THE REQUEST AND BRANCH
         BNE BADCALL                  TO ERROR ROUTINE IF NOT MQCC_OK
*
         MVC  TEMP_Q,WOD_OBJECTNAME  SAVE NAME OF TEMPORARY Q
*                            CREATED BY OPEN OF MODEL Q
*
:
BADCALL  DS   0H
```

```
⋮
*
*
*   CONSTANTS:
*
MOD_Q  DC   CL48'QUERY.REPLY.MODEL'  MODEL QUEUE NAME
DYN_Q  DC   CL48'QUERY.TEMPQ.*'      DYNAMIC QUEUE NAME
*
       CMQODA DSECT=NO,LIST=YES  CONSTANT VERSION OF MQOD
       CMQA                      MQI VALUE EQUATES
*
*   WORKING STORAGE
*
         DFHEISTG
HCONN    DS F                      CONNECTION HANDLE
OPTIONS  DS F                      OPEN OPTIONS
HOBJ     DS F                      OBJECT HANDLE
COMPCODE DS F                      MQI COMPLETION CODE
REASON   DS F                      MQI REASON CODE
TEMP_Q   DS CL(MQ_Q_NAME_LENGTH)   SAVED QNAME AFTER OPEN
*
WOD      CMQODA DSECT=NO,LIST=YES  WORKING VERSION OF MQOD
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L  LIST FORM
                                                 OF CALL
*                                                 MACRO
⋮
         END
```

Opening an existing queue

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

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

```
⋮
*
*    R5 = WORK REGISTER.
*
OPEN     DS   0H
*
         MVC  WOD_AREA,MQOD_AREA  INITIALIZE WORKING VERSION OF
*                              MQOD WITH DEFAULTS
         MVC  WOD_OBJECTNAME,Q_NAME    SPECIFY Q NAME TO OPEN
         LA   R5,MQOO_INPUT_EXCLUSIVE  OPEN FOR MQGET CALLS
*
         ST   R5,OPTIONS
*
* ISSUE MQI OPEN REQUEST USING REENTRANT FORM
* OF CALL MACRO
*
         CALL MQOPEN,                          X
               (HCONN,                         X
               WOD,                            X
               OPTIONS,                        X
               HOBJ,                           X
               COMPCODE,                       X
               REASON),VL,MF=(E,CALLLST)
*
         LA  R5,MQCC_OK         CHECK THE COMPLETION CODE
         C   R5,COMPCODE        FROM THE REQUEST AND BRANCH
         BNE BADCALL            TO ERROR ROUTINE IF NOT MQCC_OK
```

```
*
:
BADCALL  DS   0H
:
*
*
*   CONSTANTS:
*
Q_NAME   DC   CL48'REQUEST.QUEUE'  NAME OF QUEUE TO OPEN
*
         CMQODA DSECT=NO,LIST=YES  CONSTANT VERSION OF MQOD
         CMQA                      MQI VALUE EQUATES
*
*   WORKING STORAGE
*
         DFHEISTG
HCONN    DS F           CONNECTION HANDLE
OPTIONS  DS F           OPEN OPTIONS
HOBJ     DS F           OBJECT HANDLE
COMPCODE DS F           MQI COMPLETION CODE
REASON   DS F           MQI REASON CODE
*
WOD  CMQODA DSECT=NO,LIST=YES  WORKING VERSION OF MQOD
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L  LIST FORM
                                                 OF CALL
*                                                 MACRO
:
         END
```

Closing a queue

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

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

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

```
:
*                   CONSTANTS
*
         CMQA
*
*        WORKING STORAGE (REENTRANT)
*
WEG4     DSECT
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
HCONN    DS   F
HOBJ     DS   F
OPTIONS  DS   F
COMPCODE DS   F
REASON   DS   F
*
*
LEG4     EQU  *-WKEG4
         END
```

Putting a message using MQPUT

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

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

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

     LA   R5,BUFFER_LEN   RETRIEVE THE BUFFER LENGTH
     ST   R5,BUFFLEN      AND SAVE IT FOR MQM USE
*
     MVC  BUFFER,TEST_MSG    SET THE MESSAGE TO BE PUT
*
*    ISSUE MQI PUT REQUEST USING REENTRANT FORM
*    OF CALL MACRO
*
*        HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*        HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
```

```
         CALL MQPUT,                           X
               (HCONN,                         X
               HOBJ,                           X
               WMD,                            X
               WPMO,                           X
               BUFFLEN,                        X
               BUFFER,                         X
               COMPCODE,                       X
               REASON),VL,MF=(E,CALLLST)
*
         LA  R5,MQCC_OK
         C   R5,COMPCODE
         BNE BADCALL
*
.
.
.
BADCALL DS  0H
.
.
.
*
*      CONSTANTS
*
   CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
   CMQPMOA DSECT=NO,LIST=YES
   CMQA
TEST_MSG DC CL80'THIS IS A TEST MESSAGE'
*
*      WORKING STORAGE DSECT
*
WORKSTG  DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD      CMQMDA DSECT=NO,LIST=NO
WPMO     CMQPMOA DSECT=NO,LIST=NO
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
.
.
.
         END
```

Putting a message using MQPUT1

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

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

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

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

         LA   R5,BUFFER_LEN           RETRIEVE THE BUFFER LENGTH
         ST   R5,BUFFLEN              AND SAVE IT FOR MQM USE
*
         MVC  BUFFER,TEST_MSG         SET THE MESSAGE TO BE PUT
*
* ISSUE MQI PUT REQUEST USING REENTRANT FORM OF CALL MACRO
*
*        HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*        HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
         CALL MQPUT1,                                  X
                (HCONN,                                X
                 LMQOD,                                X
                 LMQMD,                                X
                 LMQPMO,                               X
                 BUFFERLENGTH,                         X
                 BUFFER,                               X
                 COMPCODE,                             X
                 REASON),VL,MF=(E,CALLLST)
*
         LA  R5,MQCC_OK
         C   R5,COMPCODE
         BNE BADCALL
*
:
BADCALL DS  0H
:
*
*      CONSTANTS
*
  CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
  CMQPMOA DSECT=NO,LIST=YES
  CMQODA DSECT=NO,LIST=YES
  CMQA
*
TEST_MSG DC CL80'THIS IS ANOTHER TEST MESSAGE'
Q_NAME   DC CL48'TEST.QUEUE.NAME'
*
*      WORKING STORAGE DSECT
*
WORKSTG  DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80
```

```
BUFFER_LEN EQU *-BUFFER
*
WOD      CMQODA DSECT=NO,LIST=YES    WORKING VERSION OF MQOD
WMD      CMQMDA DSECT=NO,LIST=NO
WPMO     CMQPMOA DSECT=NO,LIST=NO
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
⋮
         END
```

Getting a message

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

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

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

     LA   R5,BUFFER_LEN          RETRIEVE THE BUFFER LENGTH
     ST   R5,BUFFLEN             AND SAVE IT FOR MQM USE
*
*
* ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*        HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*        HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
         CALL  MQGET,                          X
               (HCONN,                         X
               HOBJ,                           X
               WMD,                            X
               WGMO,                           X
               BUFFLEN,                        X
               BUFFER,                         X
               DATALEN,                        X
               COMPCODE,                       X
               REASON),                        X
               VL,MF=(E,CALLLST)
```

```
*
         LA  R5,MQCC_OK
         C   R5,COMPCODE
         BNE BADCALL
*
:
BADCALL DS  0H
:
*
*      CONSTANTS
*
         CMQMDA DSECT=NO,LIST=YES
         CMQGMOA DSECT=NO,LIST=YES
         CMQA
*
*      WORKING STORAGE DSECT
*
WORKSTG  DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD      CMQMDA DSECT=NO,LIST=NO
WGMO     CMQGMOA DSECT=NO,LIST=NO
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
:
         END
```

Getting a message using the wait option

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

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

```
:
*     CONNECT TO QUEUE MANAGER
CONN    DS  0H
:
*     OPEN A QUEUE FOR GET
OPEN    DS  0H
:
*     R4,R5,R6,R7 = WORK REGISTER.
GET  DS  0H
     LA   R4,MQMD              SET UP ADDRESSES AND
     LA   R5,MQMD_LENGTH       LENGTH FOR USE BY MVCL
     LA   R6,WMD               INSTRUCTION, AS MQMD IS
     LA   R7,WMD_LENGTH        OVER 256 BYES LONG.
     MVCL R6,R4                INITIALIZE WORKING VERSION
*                                  OF MESSAGE DESCRIPTOR
*
     MVC  WGMO_AREA,MQGMO_AREA   INITIALIZE WORKING MQGMO
     L    R5,=AL4(MQGMO_WAIT)
```

```
     A    R5,=AL4(MQGMO_ACCEPT_TRUNCATED_MSG)
     ST   R5,WGMO_OPTIONS
     MVC  WGMO_WAITINTERVAL,TWO_MINUTES   WAIT UP TO TWO
                                          MINUTES BEFORE
                                          FAILING THE
                                          CALL
*
     LA   R5,BUFFER_LEN     RETRIEVE THE BUFFER LENGTH
     ST   R5,BUFFLEN        AND SAVE IT FOR MQM USE
*
*  ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*        HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*        HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
         CALL  MQGET,                           X
               (HCONN,                          X
               HOBJ,                            X
               WMD,                             X
               WGMO,                            X
               BUFFLEN,                         X
               BUFFER,                          X
               DATALEN,                         X
               COMPCODE,                        X
               REASON),                         X
               VL,MF=(E,CALLLST)
*
  LA  R5,MQCC_OK                  DID THE MQGET REQUEST
  C   R5,COMPCODE                   WORK OK?
  BE  GETOK                       YES, SO GO AND PROCESS.
  LA  R5,MQCC_WARNING             NO, SO CHECK FOR A WARNING.
  C   R5,COMPCODE                 IS THIS A WARNING?
  BE  CHECK_W                     YES, SO CHECK THE REASON.
*
  LA  R5,MQRC_NO_MSG_AVAILABLE   IT MUST BE AN ERROR.
                                 IS IT DUE TO AN EMPTY
  C   R5,REASON                  QUEUE?
  BE  NOMSG                      YES, SO HANDLE THE ERROR
  B   BADCALL                    NO, SO GO TO ERROR ROUTINE
*
CHECK_W  DS  0H
         LA  R5,MQRC_TRUNCATED_MSG_ACCEPTED   IS THIS A
                                              TRUNCATED
         C   R5,REASON                        MESSAGE?
         BE  GETOK                   YES, SO GO AND PROCESS.
         B   BADCALL                 NO, SOME OTHER WARNING
*
NOMSG    DS  0H
:
GETOK    DS  0H
:
BADCALL DS  0H
:
*
*      CONSTANTS
*
         CMQMDA DSECT=NO,LIST=YES
         CMQGMOA DSECT=NO,LIST=YES
         CMQA
*
TWO_MINUTES DC F'120000'       GET WAIT INTERVAL
*
*      WORKING STORAGE DSECT
*
WORKSTG  DSECT
*
```

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

Getting a message using signaling

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

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

```
.
.
.
*
*     CONNECT TO QUEUE MANAGER
*
CONN    DS  0H
.
.
.
*
*     OPEN A QUEUE FOR GET
*
OPEN    DS  0H
.
.
.
*
*     R4,R5,R6,R7 = WORK REGISTER.
*
GET  DS  0H
     LA   R4,MQMD                SET UP ADDRESSES AND
     LA   R5,MQMD_LENGTH         LENGTH FOR USE BY MVCL
     LA   R6,WMD                 INSTRUCTION, AS MQMD IS
     LA   R7,WMD_LENGTH          OVER 256 BYES LONG.
     MVCL R6,R4                  INITIALIZE WORKING VERSION
*                                   OF MESSAGE DESCRIPTOR
*
     MVC  WGMO_AREA,MQGMO_AREA   INITIALIZE WORKING MQGMO
     LA   R5,MQGMO_SET_SIGNAL
     ST   R5,WGMO_OPTIONS
     MVC  WGMO_WAITINTERVAL,FIVE_MINUTES   WAIT UP TO FIVE
                                           MINUTES BEFORE
*                                          FAILING THE CALL
*
     XC   SIG_ECB,SIG_ECB   CLEAR THE ECB
     LA   R5,SIG_ECB        GET THE ADDRESS OF THE ECB
     ST   R5,WGMO_SIGNAL1   AND PUT IT IN THE WORKING
*                                  MQGMO
*

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

```
*
*
*    ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*        HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*        HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
         CALL  MQGET,                           X
               (HCONN,                          X
               HOBJ,                            X
               WMD,                             X
               WGMO,                            X
               BUFFLEN,                         X
               BUFFER,                          X
               DATALEN,                         X
               COMPCODE,                        X
               REASON),                         X
               VL,MF=(E,CALLLST)
*
         LA  R5,MQCC_OK        DID THE MQGET REQUEST
         C   R5,COMPCODE       WORK OK?
         BE  GETOK             YES, SO GO AND PROCESS.
         LA  R5,MQCC_WARNING   NO, SO CHECK FOR A WARNING.
         C   R5,COMPCODE       IS THIS A WARNING?
         BE  CHECK_W           YES, SO CHECK THE REASON.
         B   BADCALL           NO, SO GO TO ERROR ROUTINE
*
CHECK_W  DS  0H
         LA  R5,MQRC_SIGNAL_REQUEST_ACCEPTED
         C   R5,REASON   SIGNAL REQUEST SIGNAL SET?
         BNE BADCALL     NO, SOME ERROR OCCURRED
         B   DOWORK      YES, SO DO SOMETHING
*                          ELSE
*
CHECKSIG DS  0H
         CLC SIG_ECB+1(3),=AL3(MQEC_MSG_ARRIVED)
                              IS A MESSAGE AVAILABLE?
         BE  GET                YES, SO GO AND GET IT
*
         CLC SIG_ECB+1(3),=AL3(MQEC_WAIT_INTERVAL_EXPIRED)
                           HAVE WE WAITED LONG ENOUGH?
         BE  NOMSG         YES, SO SAY NO MSG AVAILABLE
         B   BADCALL       IF IT'S ANYTHING ELSE
*                          GO TO ERROR ROUTINE.
*
DOWORK   DS  0H
:
         TM  SIG_ECB,X'40'   HAS THE SIGNAL ECB BEEN POSTED?
         BO  CHECKSIG        YES, SO GO AND CHECK WHY
         B   DOWORK          NO, SO GO AND DO MORE WORK
*
NOMSG    DS  0H
:
GETOK    DS  0H
:
BADCALL DS  0H
:
*
*      CONSTANTS
*
         CMQMDA DSECT=NO,LIST=YES
         CMQGMOA DSECT=NO,LIST=YES
         CMQA
*
FIVE_MINUTES DC F'300000'        GET SIGNAL INTERVAL
*
```

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

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

Inquiring about and setting the attributes of a queue

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

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

```
.
.
.
DFHEISTG DSECT
.
.
.
OBJDESC  CMQODA LIST=YES   Working object descriptor
*
SELECTORCOUNT   DS F       Number of selectors
INTATTRCOUNT    DS F       Number of integer attributes
CHARATTRLENGTH  DS F       char attributes length
CHARATTRS       DS C       Area for char attributes
*
OPTIONS  DS   F            Command options
HCONN    DS   F            Handle of connection
HOBJ     DS   F            Handle of object
COMPCODE DS   F            Completion code
REASON   DS   F            Reason code
SELECTOR DS   2F           Array of selectors
INTATTRS DS   2F           Array of integer attributes
.
.
.
OBJECT   DS   CL(MQ_Q_NAME_LENGTH)   Name of queue
.
.
.
CALLLIST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
************************************************************
*                   PROGRAM EXECUTION STARTS HERE        *
.
.
.
CSQ4CAC1 DFHEIENT CODEREG=(R3),DATAREG=(R13)
.
.
.
*        Initialize the variables for the set call
*
         SR   R0,R0              Clear register zero
```

```
         ST   R0,CHARATTRLENGTH   Set char length to zero
         LA   R0,2                Load to set
         ST   R0,SELECTORCOUNT    selectors add
         ST   R0,INTATTRCOUNT     integer attributes
*
         LA   R0,MQIA_INHIBIT_GET Load q attribute selector
         ST   R0,SELECTOR+0       Place in field
         LA   R0,MQIA_INHIBIT_PUT Load q attribute selector
         ST   R0,SELECTOR+4       Place in field
*
UPDTEST  DS   0H
         CLC  ACTION,CINHIB       Are we inhibiting?
         BE   UPDINHBT            Yes branch to section
*
         CLC  ACTION,CALLOW       Are we allowing?
         BE   UPDALLOW            Yes branch to section
*
         MVC  M00_MSG,M01_MSG1    Invalid request
         BR   R6                  Return to caller
*
UPDINHBT DS   0H
         MVC  UPDTYPE,CINHIBIT      Indicate action type
         LA   R0,MQQA_GET_INHIBITED Load attribute value
         ST   R0,INTATTRS+0         Place in field
         LA   R0,MQQA_PUT_INHIBITED Load attribute value
         ST   R0,INTATTRS+4         Place in field
         B    UPDCALL               Go and do call
*
UPDALLOW DS   0H
         MVC  UPDTYPE,CALLOWED      Indicate action type
         LA   R0,MQQA_GET_ALLOWED   Load attribute value
         ST   R0,INTATTRS+0         Place in field
         LA   R0,MQQA_PUT_ALLOWED   Load attribute value
         ST   R0,INTATTRS+4         Place in field
         B    UPDCALL               Go and do call
*
UPDCALL  DS   0H
         CALL MQSET,                          C
               (HCONN,                        C
               HOBJ,                          C
               SELECTORCOUNT,                 C
               SELECTOR,                      C
               INTATTRCOUNT,                  C
               INTATTRS,                      C
               CHARATTRLENGTH,                C
               CHARATTRS,                     C
               COMPCODE,                      C
               REASON),                       C
               VL,MF=(E,CALLLIST)
*
         LA   R0,MQCC_OK    Load expected compcode
         C    R0,COMPCODE   Was set successful?
:
*  SECTION NAME : INQUIRE                                *
*  FUNCTION     : Inquires on the objects attributes     *
*  CALLED BY    : PROCESS                                *
*  CALLS        : OPEN, CLOSE, CODES                     *
*  RETURN       : To Register 6                          *
INQUIRE  DS   0H
:

*        Initialize the variables for the inquire call
*
         SR   R0,R0               Clear register zero
         ST   R0,CHARATTRLENGTH   Set char length to zero
         LA   R0,2                Load to set
```

```
         ST   R0,SELECTORCOUNT    selectors add
         ST   R0,INTATTRCOUNT     integer attributes
*
         LA   R0,MQIA_INHIBIT_GET Load attribute value
         ST   R0,SELECTOR+0       Place in field
         LA   R0,MQIA_INHIBIT_PUT Load attribute value
         ST   R0,SELECTOR+4       Place in field
         CALL MQINQ,                          C
               (HCONN,                        C
               HOBJ,                          C
               SELECTORCOUNT,                 C
               SELECTOR,                      C
               INTATTRCOUNT,                  C
               INTATTRS,                      C
               CHARATTRLENGTH,                C
               CHARATTRS,                     C
               COMPCODE,                      C
               REASON),                       C
               VL,MF=(E,CALLLIST)
         LA   R0,MQCC_OK        Load expected compcode
         C    R0,COMPCODE       Was inquire successful?
:
```

Chapter 8. PL/I examples

The use of PL/I is supported by z/OS only.

The examples demonstrate the following techniques:

Connecting to a queue manager

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

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

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

```
                HCONN,       /* CONNECTION HANDLE      */
                COMPCODE,    /* COMPLETION CODE        */
                REASON);     /* REASON CODE            */

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

Disconnecting from a queue manager

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

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

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

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

Creating a dynamic queue

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

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

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

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

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

:
          CALL ERROR_ROUTINE;
        END;
        ELSE
          DYNAMIC_QUEUE_NAME = LMQOD_OBJECTNAME;
```

Opening an existing queue

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

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

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
```

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

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

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

:
          CALL ERROR_ROUTINE;
        END;
```

Closing a queue

This example demonstrates how to use the MQCLOSE call.

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

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

```
OPTIONS=MQCO_NONE;

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

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

⋮
          CALL ERROR_ROUTINE;
        END;
```

Putting a message using MQPUT

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

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

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

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

/*******************************************************/
/* SET UP LENGTH OF MESSAGE BUFFER AND THE MESSAGE     */
/*******************************************************/
BUFFLEN = LENGTH(BUFFER);
BUFFER = PL1_TEST_MESSAGE;
/*******************************************************/
/*                                                     */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.           */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.            */
/*                                                     */
/*******************************************************/
CALL MQPUT (HCONN,
            HOBJ,
            LMQMD,
            LMQPMO,
            BUFFLEN,
            BUFFER,
            COMPCODE,
            REASON);
/*******************************************************/
/* TEST THE COMPLETION CODE OF THE PUT CALL.           */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE       */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.    */
/*******************************************************/
    IF COMPCODE ¬= MQCC_OK
       THEN DO;

:
         CALL ERROR_ROUTINE;
       END;
```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call.

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

```
%INCLUDE SYSLIB(CMQEPP);
%INCLUDE SYSLIB(CMQP);
:
/*******************************************************/
/* WORKING STORAGE DECLARATIONS                        */
/*******************************************************/
DCL COMPCODE            BINARY FIXED (31);
DCL REASON              BINARY FIXED (31);
DCL HCONN               BINARY FIXED (31);
DCL OPTIONS             BINARY FIXED (31);
DCL BUFFLEN             BINARY FIXED (31);
DCL BUFFER              CHAR(80);
:
DCL REPLY_TO_QUEUE      CHAR(48) INIT('PL1.REPLY.QUEUE');
DCL QUEUE_NAME          CHAR(48) INIT('PL1.LOCAL.QUEUE');
DCL PL1_TEST_MESSAGE    CHAR(80)
  INIT('***** THIS IS ANOTHER TEST MESSAGE *****');
:
/*******************************************************/
/* LOCAL COPY OF OBJECT DESCRIPTOR, MESSAGE DESCRIPTOR   */
```

```
/* AND PUT MESSAGE OPTIONS                                  */
/************************************************************/
DCL 1 LMQOD  LIKE MQOD;
DCL 1 LMQMD  LIKE MQMD;
DCL 1 LMQPMO LIKE MQPMO;
⋮
/************************************************************/
/* SET UP OBJECT DESCRIPTOR AS REQUIRED.                    */
/************************************************************/
LMQOD.OBJECTTYPE = MQOT_Q;
LMQOD.OBJECTNAME = QUEUE_NAME;

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

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

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

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

⋮
          CALL ERROR_ROUTINE;
        END;
```

Getting a message

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

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

```
     %INCLUDE SYSLIB(CMQP);
     %INCLUDE SYSLIB(CMQEPP);
     :
```

```
/*************************************************************/
/* WORKING STORAGE DECLARATIONS                              */
/*************************************************************/
     DCL COMPCODE                 BINARY FIXED (31);
     DCL REASON                   BINARY FIXED (31);
     DCL HCONN                    BINARY FIXED (31);
     DCL HOBJ                     BINARY FIXED (31);
     DCL BUFFLEN                  BINARY FIXED (31);
     DCL DATALEN                  BINARY FIXED (31);
     DCL BUFFER                   CHAR(80);

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

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

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

/*************************************************************/
/* SET UP LENGTH OF MESSAGE BUFFER.                          */
/*************************************************************/
     BUFFLEN = LENGTH(BUFFER);
/*************************************************************/
/*                                                           */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.                 */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.                  */
/*                                                           */
/*************************************************************/

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

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

Getting a message using the wait option

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

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

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

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

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

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

/*******************************************************/
/* SET UP LENGTH OF MESSAGE BUFFER.                    */
/*******************************************************/
     BUFFLEN = LENGTH(BUFFER);
/*******************************************************/
/*                                                     */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.           */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.            */
/*                                                     */
/*******************************************************/

     CALL MQGET (HCONN,
                 HOBJ,
                 LMQMD,
                 LMQGMO,
                 BUFFERLEN,
                 BUFFER,
                 DATALEN,
```

```
                COMPCODE,
                REASON);

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

     SELECT(COMPCODE);
       WHEN (MQCC_OK) DO;   /* GET WAS SUCCESSFUL */

.
.
.
       END;
       WHEN (MQCC_WARNING) DO;
         IF REASON = MQRC_TRUNCATED_MSG_ACCEPTED
           THEN DO;         /* GET WAS SUCCESSFUL */

.
.
.
           END;
           ELSE DO;

.
.
.
             CALL ERROR_ROUTINE;
           END;
       END;
       WHEN (MQCC_FAILED) DO;

.
.
.
           CALL ERROR_ROUTINE;
         END;
       END;
       OTHERWISE;
     END;
```

Getting a message using signaling

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

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

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

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

.
.
.
     DCL ECB_FIXED           FIXED BIN(31);
     DCL 1 ECB_OVERLAY BASED(ADDR(ECB_FIXED)),
           3 ECB_WAIT    BIT,
           3 ECB_POSTED  BIT,
           3 ECB_FLAG3_8 BIT(6),
```

```
          3 ECB_CODE  PIC'999';

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

.
.
.
/*********************************************************/
/* CLEAR ECB FIELD.                                      */
/*********************************************************/
     ECB_FIXED = 0;

.
.
.
/*********************************************************/
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED.                */
/* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST      */
/* AVAILABLE MESSAGE WILL BE RETRIEVED.                  */
/*********************************************************/
     LMQMD.MSGID = MQMI_NONE;
     LMQMD.CORRELID = MQCI_NONE;
/*********************************************************/
/* SET UP GET MESSAGE OPTIONS AS REQUIRED.               */
/* WAIT INTERVAL SET TO ONE MINUTE.                      */
/*********************************************************/
     LMQGMO.OPTIONS = MQGMO_SET_SIGNAL +
                      MQGMO_NO_SYNCPOINT;
     LMQGMO.WAITINTERVAL=60000;
     LMQGMO.SIGNAL1 = ADDR(ECB_FIXED);
/*********************************************************/
/* SET UP LENGTH OF MESSAGE BUFFER.                      */
/* CALL MESSGE RETRIEVAL ROUTINE.                        */
/*********************************************************/
     BUFFLEN = LENGTH(BUFFER);
     CALL GET_MSG;

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

     SELECT;
       WHEN ((COMPCODE = MQCC_OK) &
             (REASON = MQCC_NONE)) DO

.
.
.
         CALL MSG_ROUTINE;

.
.
.
       END;
       WHEN ((COMPCODE = MQCC_WARNING) &
             (REASON = MQRC_SIGNAL_REQUEST_ACCEPTED)) DO;

.
.
.
         CALL DO_WORK;

.
.
.
       END;
       WHEN  ((COMPCODE = MQCC_FAILED) &
             (REASON = MQRC_SIGNAL_OUTSTANDING)) DO;
```

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

          END;

        END;
.
.
.
 END DO_WORK;

 GET_MSG: PROC;

/*******************************************************/
/*                                                     */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.           */
```

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

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

 END GET_MSG;

 NO_MSG: PROC;

⋮
 END NO_MSG;
```

Inquiring about the attributes of an object

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

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

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

⋮

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

     SELECTORS(01) = MQIA_CURRENT_Q_DEPTH;

     SELECTORCOUNT  = 1;
     INTATTRCOUNT   = 1;

     CHARATTRLENGTH = 0;
/******************************************************/
/*                                                    */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.          */
```

```
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.             */
/*                                                      */
/********************************************************/
     CALL MQINQ (HCONN,
                 HOBJ,
                 SELECTORCOUNT,
                 SELECTORS,
                 INTATTRCOUNT,
                 INTATTRS,
                 CHARATTRLENGTH,
                 CHARATTRS,
                 COMPCODE,
                 REASON);
/********************************************************/
/* TEST THE COMPLETION CODE OF THE INQUIRE CALL.        */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING */
/* THE COMPLETION CODE AND THE REASON CODE.             */
/********************************************************/
     IF COMPCODE ¬= MQCC_OK
        THEN DO;

⋮
          CALL ERROR_ROUTINE;
        END;
```

Setting the attributes of a queue

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

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

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

⋮

/********************************************************/
/* SET VARIABLES FOR SET CALL                           */
/* SET GET AND PUT INHIBITED                            */
/********************************************************/

     SELECTORS(01) = MQIA_INHIBIT_GET;
     SELECTORS(02) = MQIA_INHIBIT_PUT;

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

```
     SELECTORCOUNT  = 2;
     INTATTRCOUNT   = 2;

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

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

.
.
.
          CALL ERROR_ROUTINE;
        END;
```

Chapter 9. WebSphere MQ data definition files

WebSphere MQ provides data definition files to help you to write your applications. Data definition files are also known as:

Language	Data definitions
C	Include files or header files
Visual Basic	Module files (32-bit versions only)
COBOL	Copy files
Assembler	Macros
PL/I	Include files

The data definition files to help you to write channel exits are described in WebSphere MQ Intercommunication.

The data definition files to help you to write installable services exits are described in the WebSphere MQ System Administration Guide.

For data definition files supported on C++, see WebSphere MQ Using C++.

For data definition files supported on RPG, see the WebSphere MQ for i5/OS Application Programming Reference (ILE/RPG).

The names of the data definition files have the prefix CMQ, and a suffix that is determined by the programming language:

Suffix	Language
a	Assembler language
b	Visual Basic
c	C
l	COBOL (without initialized values)
p	PL/I
v	COBOL (with default values set)

Installation library

The name **thlqual** is the high-level qualifier of the installation library on z/OS.

This chapter introduces WebSphere MQ data definition files, under these headings:

C language include files

The WebSphere MQ C include files are listed in WebSphere MQ Constants. They are installed in the following directories or libraries:

Platform	Installation directory or library
AIX	/usr/mqm/inc/
i5/OS	QMQM/H
UNIX platforms	/opt/mqm/inc/
Windows systems	\Program Files\IBM\WebSphere MQ\Tools\c\include
z/OS	**thlqual**.SCSQC370

Note: For UNIX platforms, the include files are symbolically linked into `/usr/include`.

For more information on the structure of directories, see the WebSphere MQ System Administration Guide.

Visual Basic module files

WebSphere MQ for Windows provides four Visual Basic module files.

They are listed in WebSphere MQ Constants and installed in

```
\Program Files\IBM\WebSphere MQ\Tools\Samples\VB\Include
```

COBOL copy files

For COBOL, WebSphere MQ provides separate copy files containing the named constants, and two copy files for each of the structures.

There are two copy files for each structure because each is provided both with and without initial values:

- In the WORKING-STORAGE SECTION of a COBOL program, use the files that initialize the structure fields to default values. These structures are defined in the copy files that have names suffixed with the letter V (values).
- In the LINKAGE SECTION of a COBOL program, use the structures without initial values. These structures are defined in copy files that have names suffixed with the letter L (linkage).

Copy files containing data and interface definitions for WebSphere MQ for i5/OS are provided for ILE COBOL programs using prototyped calls to the MQI. The files exist in QMQM/QCBLLESRC with member names that have a suffix of L (for structures without initial values) or a suffix of V (for structures with initial values).

The WebSphere MQ COBOL copy files are listed in WebSphere MQ Constants. They are installed in the following directories:

Platform	Installation directory or library
AIX	/usr/mqm/inc/
Other UNIX platforms	/opt/mqm/inc/
i5/OS	QMQM/QCBLLESRC

Platform	Installation directory or library
Windows	\Program Files\IBM\WebSphere MQ\Tools\cobol\ copybook (for Micro Focus COBOL) \Program Files\IBM\WebSphere MQ\Tools\cobol\copybook\VAcobol (for IBM VisualAge COBOL)
z/OS	**thlqual**.SCSQCOBC

Include in your program only those files that you need. Do this with one or more COPY statements after a level-01 declaration. This means that you can include multiple versions of the structures in a program if necessary. Note that CMQV is a large file.

Here is an example of COBOL code to include the CMQMDV copy file:

```
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
```

Each structure declaration begins with a level-01 item; you can declare several instances of the structure by coding the level-01 declaration followed by a COPY statement to copy in the remainder of the structure declaration. To refer to the appropriate instance, use the IN keyword.

Here is an example of COBOL code to include two instances of CMQMDV:

```
* Declare two instances of MQMD
 01  MY-CMQMD.
     COPY CMQMDV.
 01  MY-OTHER-CMQMD.
     COPY CMQMDV.
*
* Set MSGTYPE field in MY-OTHER-CMQMD
     MOVE MQMT-REQUEST TO MQMD-MSGTYPE IN MY-OTHER-CMQMD.
```

Align the structures on 4-byte boundaries. If you use the COPY statement to include a structure following an item that is not the level-01 item, ensure that the structure is a multiple of 4-bytes from the start of the level-01 item. If you do not do this, you might reduce the performance of your application.

The structures are described in the WebSphere MQ Application Programming Reference. The descriptions of the fields in the structures show the names of fields without a prefix. In COBOL programs, prefix the field names with the name of the structure followed by a hyphen, as shown in the COBOL declarations. The fields in the structure copy files are prefixed in this way.

The field names in the declarations in the structure copy files are in uppercase. You can use mixed case or lowercase instead. For example, the field *StrucId* of the MQGMO structure is shown as MQGMO-STRUCID in the COBOL declaration and in the copy file.

The V-suffix structures are declared with initial values for all the fields, so you need to set only those fields where the value required is different from the initial value.

System/390 assembler-language macros

WebSphere MQ for z/OS provides two assembler-language macros containing the named constants, and one macro to generate each structure.

They are listed in WebSphere MQ Constants and installed in **thlqual**.SCSQMACS.

These macros are called using code like this:

```
MY_MQMD  CMQMDA EXPIRY=0,MSGTYPE=MQMT_DATAGRAM
```

PL/I include files

WebSphere MQ for z/OS provides include files that contain all the definitions that you need when you write WebSphere MQ applications in PL/I.

The files are listed in WebSphere MQ Constants and installed in the **thlqual**.SCSQPLIC directory:

Include these files in your program if you are going to link the WebSphere MQ stub to your program (see "Preparing your program to run" on page 373). Include only CMQP if you intend to link the WebSphere MQ calls dynamically (see "Dynamically calling the WebSphere MQ stub" on page 377). Dynamic linking can be performed for batch and IMS programs only.

Chapter 10. Coding standards on 64 bit platforms

Preferred data types

These types never change size and are available on both 32-bit and 64-bit WebSphere MQ platforms:

Name	Length
MQLONG	4 bytes
MQULONG	4 bytes
MQINT32	4 bytes
MQUINT32	4 bytes
MQINT64	8 bytes
MQUINT64	8 bytes

Standard data types

32-bit UNIX applications

This section is included for comparison and is based on Solaris. Any differences with other UNIX platforms are noted:

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	16 bytes Note that on AIX and Linux PPC a long double is 8 bytes.
pointer	4 bytes
ptrdiff_t	4 bytes
size_t	4 bytes
time_t	4 bytes
clock_t	4 bytes
wchar_t	4 bytes Note that on AIX a wchar_t is 2 bytes.

64-bit UNIX applications

This section is based on Solaris. Any differences with other UNIX platforms are noted:

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes

Name	Length
float	4 bytes
double	8 bytes
long double	16 bytes
	Note that on AIX and Linux PPC a long double is 8 bytes.
pointer	8 bytes
ptrdiff_t	8 bytes
size_t	8 bytes
time_t	8 bytes
clock_t	8 bytes
	Note that on the other UNIX platforms a clock_t is 4 bytes.
wchar_t	4 bytes
	Note that on AIX a wchar_t is 2 bytes.

Windows 64–bit applications

Name	Length
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes
pointer	8 bytes
	Note that all pointers are 8 bytes.
ptrdiff_t	8 bytes
size_t	8 bytes
time_t	8 bytes
clock_t	4 bytes
wchar_t	2 bytes
WORD	2 bytes
DWORD	4 bytes
HANDLE	8 bytes
HFILE	4 bytes

Coding considerations on Windows

HANDLE hf;:

Use

```
hf = CreateFile((LPCTSTR) FileName,
                Access,
                ShareMode,
                xihSecAttsNTRestrict,
                Create,
                AttrAndFlags,
                NULL);
```

Do not use

```
HFILE hf;
hf = (HFILE) CreateFile((LPCTSTR) FileName,
                        Access,
```

```
                        ShareMode,
                        xihSecAttsNTRestrict,
                        Create,
                        AttrAndFlags,
                        NULL);
```

as this produces an error.

size_t len fgets:

Use

```
size_t len
while (fgets(string1, (int) len, fp) != NULL)
len = strlen(buffer);
```

Do not use

```
int len;

while (fgets(string1, len, fp) != NULL)
len = strlen(buffer);
```

printf:

Use

```
printf("My struc pointer:
```

Do not use

```
printf("My struc pointer:
```

If you need hexadecimal output, you have to print the upper and lower 4 bytes separately.

char *ptr:

Use

```
char * ptr1;
char * ptr2;
size_t bufLen;

bufLen = ptr2 - ptr1;
```

Do not use

```
char *ptr1;
char *ptr2;
UINT32 bufLen;

bufLen = ptr2 - ptr1;
```

alignBytes:

Use

```
alignBytes = (unsigned short) ((size_t) address
```

Do not use

```
void *address;
unsigned short alignBytes;

alignBytes = (unsigned short) ((UINT32) address
```

alignBytes:

Use

```
alignBytes = (unsigned short) ((size_t) address
```

Do not use

```
void *address;
unsigned short alignBytes;

alignBytes = (unsigned short) ((UINT32) address
```

len:

Use

```
len = (UINT32) ((char *) address2 - (char *) address1);
```

Do not use

```
void *address1;
void *address2;
UINT32 len;

len = (UINT32) ((char *) address2 - (char *) address1);
```

sscanf:

Use

```
MQLONG SBCSprt;

sscanf(line, "SBCSprt);
```

Do not use

```
MQLONG SBCSprt;

sscanf(line, "SBCSprt);
```

`%ld` tries to put an eight byte type into a four byte type; only use `%l` if you are dealing with an actual `long` data type. MQLONG, UINT32 and INT32 are defined to be four bytes, the same as an `int` on all WebSphere MQ platforms:

Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing,
IBM Corporation,
North Castle Drive,
Armonk, NY 10504-1785,
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation,
Licensing,
2-31 Roppongi 3-chome, Minato-k,u
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	C/370	C/400
CICS	CICS/ESA®	COBOL/400
Common User Access®	DB2	Encina
FFST	i5/OS	IBM
IMS	IMS/ESA	Informix
Integrated Language Environment	Intel	Itanium®
Language Environment	Lotus Notes	MVS
Notes	OS/390	RACF
Redbooks	SAA®	SupportPac
System i	System/390®	TXSeries

UNIX	VisualAge	VM/ESA
VSE/ESA	WebSphere	z/OS
zSeries®		

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44-1962-816151
  - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

SC34-6939-00

Spine information:

WebSphere MQ Application Programming Guide Version 7.0